Code Splitting for Network Bound Web 2.0 Applications

Benjamin Livshits[†] *Chen Ding*[†,⋆]

Microsoft Research[†] *University of Rochester*[⋆]

Abstract

Modern Web 2.0 applications such as Gmail, Live Maps, MySpace, Flickr and many others have become a common part of everyday life. These applications are network bound, meaning that their performance can and does vary a great deal based on network conditions. However, there has not been much systematic research on trying to optimize network usage of these applications to make them more responsive for end-user interactions.

Interestingly enough, code itself, usually in the form of JavaScript that runs within the client browser, constitutes a significant fraction of what needs to be transferred to the client for the application to run. Therefore, one way to significantly improve the perceived client-side performance for a range of Web 2.0 applications is to perform judicious decomposition — or *splitting* — of code and only transfer the code that is necessary just before it is needed. In this paper we propose several code splitting algorithms and explore their effectiveness at improving the responsiveness of large sophisticated Web 2.0 sites.

1 Introduction

Web 2.0 — an emerging phenomenon just a few years ago — has arrived. Today it is often hard to imagine one's daily existence without Gmail, Live Maps, RedFin, MySpace, NetFlix, and other similarly ubiquitous applications. All of these are Web 2.0 applications enabled by better user experience provided through the Ajax (Asynchronous JavaScript and XML) set of technologies. It is likely that the development of such applications will continue, fueled both by start-ups and large companies competing for customer attention.

From the technical standpoint, a key distinguishing characteristic of Web 2.0 applications is the fact that code executes both on the client, within the Web browser, and on the server, whose capacity ranges from a standalone machine to a full-fledged data center. Simply put, today's Web 2.0 applications are effectively sophisticated distributed systems, with the client portion typically written in JavaScript running within the browser. Client-side execution leads to faster, more *responsive client-side experience*, which makes Web 2.0 sites shine compared to their "traditional" Web 1.0 application counterparts.

In traditional Web applications execution occurs *entirely* on the server so that every client-side update within the browser triggers a round-trip back to the server, followed by a refresh of the *entire* browser window. In contrast, Web 2.0 applications make requests to fetch only the data that is necessary and are able to repaint individual portions of the screen. For instance, a mapping applications such as Google Maps or Live Maps may only fetch map tiles around a particular point of interest such as a street address or a landmark. Once additional bandwidth becomes available, such an application may use speculative data prefetch: it could push additional map tiles for the surrounding regions of the map. This is beneficial because if the user chooses to move the map around, as supported by interactive Ajax experience, surrounding tiles will already be available on the client side in the browser cache.

However, there is an even more basic bottleneck associated with today's sophisticated Web 2.0 applications: they contain a great deal of code. For large applications, downloading as much as 1 MB of JavaScript code on the first visit to the front page is not at all uncommon. However, since much of application execution occurs on the client, the code must be transferred for that execution to proceed. 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 user-friendly experience, especially on slower connections. Indeed, on a 56K modem, on even the second load of large applications like Live Maps becomes virtually unusable, taking over 3 minutes to load. In the international context there are many users that only have access to low-bandwidth connections and improving server-side infrastructure will still not place Web 2.0 within their reach. Even on a typical wireless connection the simple act of opening an email from an inbox can take 24 seconds on the first Hotmail visit and 11 seconds subsequently. Even small degradation in the execution time of such common tasks may significantly impact the end-user experience.

In this paper we explore the benefits of several code splitting schemes we have developed for Web 2.0 applications. We perform an evaluation based on instrumented execution traces we have collected and we estimate the benefits of code splitting techniques for a variety of network conditions, including 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

Web application	Site URL	Workload performed	Download size in KB	Waiting time, in seconds					
				56k modem		802.11b wireless		Cable modem	
Gmail	`mail.google.com`	Open 1^{st} email	249	1:07.5	16.0	18.0	9.8	9.1	8.8
Bunny Hunt	`www.themaninblue.com/...`[1]	Load page	274	55.6	42.4	10.0	7.5	2.3	1.8
Dropthings	`www.dropthings.com`	Load page	520	1:24.4	31.8	29.0	1.8	5.1	1.3
Google Maps	`maps.google.com`	Load page	716	1:09.4	15.2	18.8	4.0	4.7	2.3
Live.com	`www.live.com`	Load page	1,030	4:10.6	19.8	45.7	6.3	8.2	5.0
Pageflakes	`www.pageflakes.com`	Load page	1,067	4:07.4	28.5	1:25.1	18.6	16.4	9.8
Hotmail	`www.hotmail.com`	Open 1^{st} email	1,239	3:02.0	3:06.1	24.0	11.2	8.3	6.1
Live Maps	`maps.live.com`	Load page	1,250	3:51.3	18.5	47.9	4.8	5.2	3.2

Figure 1: Summary of information about some widely used Web 2.0 applications, sorted by download size

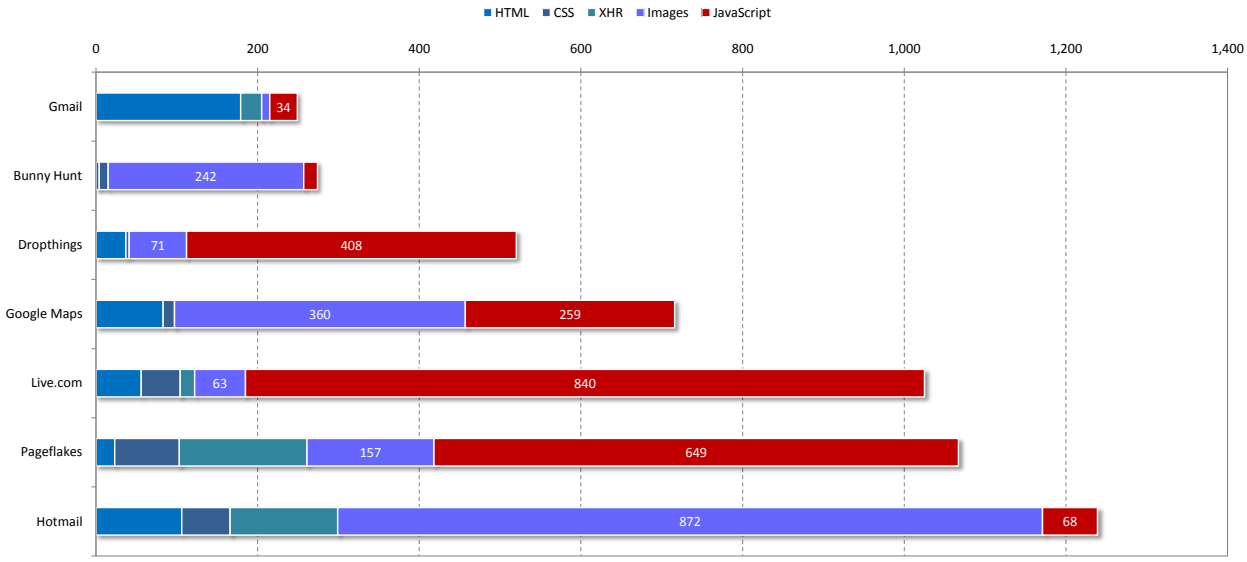

Figure 2: Size breakdown of different Web 2.0 application components

if transferring the entire code.

1.1 Contributions

This paper makes the following contributions:

- We propose code splitting as a means to improve the perceived responsiveness of Web 2.0 applications within the user's browser.
- We describe some typical architectural patterns pertaining to Web 2.0 applications and demonstrate how code splitting can greatly increase application responsiveness. Based on our insight, we propose four different code splitting schemes that are likely to speed-up a range of Web 2.0 applications.
- We develop a formal model for reasoning about and analyzing the waiting time for each of the code splitting approaches for a range of network settings.
- Using our model, we evaluate the effectiveness of various code splitting strategies for a range of real usage traces of popular Web 2.0 applications.

1.2 Paper Organization

The rest of the paper is organized as follows. Section 2 gives an overview of common application construction patterns and of why code splitting is a good strategy for improving application responsiveness. Section 3 presents a formal model that defines an optimization problem we are solving as well as several code splitting strategies. Section 4 shows potential improvements of various code splitting schemes for our benchmark applications. Finally, Sections 5 and 6 describe related work and conclude.

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

Figure 3: A possible script loading strategy from the Bunny Hunt JavaScript game (`http://www.themaninblue.com/experiment/BunnyHunt`)

2 Overview

In this section we examine some of the issues that arise in the context of developing Web 2.0 applications through examining several existing large applications.

2.1 Benchmarks Applications

A summary of information about our benchmark applications is given in Figure 1. To measure the size of the download for each application, we started by clearing the browser cache and then visiting the site and performing the action specified in the figure. For most sites, it involved just loading the page, which in turn triggered download of both static content such as HTML and image files as well as JavaScript code. In the case of two Webmail applications, our workload consisted of (1) opening the inbox and then (2) opening the first email. We waited for all images and ads to download at intermediate steps of this process. Furthermore, to avoid timing inconsistencies associated with manual typing, we have an authentication cookie that obviated the need for manual login. To create a realistic workload scenarios for the mapping applications, we waited until the main page finished loading, entered a city name, waited until the city map finished loading, and double-clicked on the map.

It is worth pointing out that this initial download size as specified in Figure 1 is by no means final: in the course of application usage, there is certainly more and more data that is downloaded into the user's browser. However, it has been our experience that the *majority* of code is downloaded on the initial page visit. For some applications such as Live Maps, there is clearly an attempt to perform code decomposition. For example, when browsing the downloaded map and asking for traffic information for a particular city, new snippets of JavaScript that correspond to that particular city would be downloaded to the client. Despite that fact, among our benchmarks, Live Maps has by far the biggest initial JavaScript download exceeding 900 KB.

The right portion of the table in Figure 1 summarizes the execution times for our workload to perform each action for different connection speed, both with a clear browser cache and with the "pre-seeded" cache on the second visit. On a typical modem connection, execution speed become on the order of several minutes, rendering these applications virtually unusable. In most cases, the effects of client-side caching are quite pronounced: for example, on a second visit of Live Maps on a modem connection, the execution time is cut by 92%. A notable exception is Bunny Hunt, where the drop in execution time is quite insignificant. This is because Bunny Hunt uses dynamic HTML rewriting to download images to be loaded and these dynamically loaded images are not added to the browser cache.

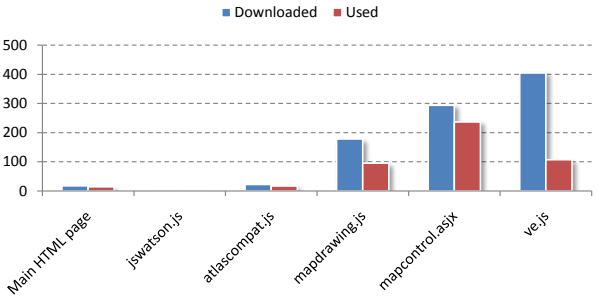

Figure 4: Sizes of JavaScript files for Live Maps, in KB

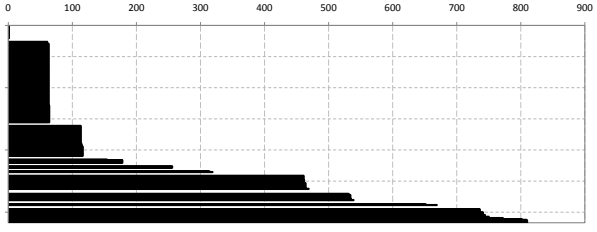

Figure 5: Difference between the available and demand time for functions in `mapcontrol.asjx`, a 291 KB file on Live Maps. Functions that are never called are not shown. Gaps are inserted to show the natural block structure.

2.2 Typical Code Decomposition Scenarios

The most natural way to transfer JavaScript files or, indeed, any type of resource is by specifying their names directly in HTML, as shown in Figure 3. This is not the only approach to resource loading: indeed resources can be loaded dynamically using `document.write` [6]. One advantage to static resource loading is that the order in which they are processed by the browser is guaranteed, whereas in the case of dynamic resource loading, additional code needs to be added to make sure that a JavaScript file is available on the client before a function from it is called. A disadvantage is that no parallel or out-of-order loading is possible.

It is often recommended to break down the application so that only a small portion of the code — the framework — is transferred to the client initially. Subsequent code can be transferred whenever bandwidth becomes available. Alternatively, it could be transferred in a context-sensitive manner. For instance, "help" functionality is used very rarely, the JavaScript code behind it can be transferred upon the help button being pressed. Similarly, if reading an Ajax-enhanced blog is much more common than posting to it, JavaScript responsible for posting can be downloaded later on demand.

To reduce the waiting time and improve the user experience, Web 2.0 application developers typically decompose their code into logical units. However, often the break-down is accomplished in an ad-hoc fashion that is optimized for the ease of use for the developer instead of the responsiveness of the application for the end-user. To get a better sense for the decomposition decisions developers make, it is instructive to look at some typical application decomposition scenarios for some representative existing applications.

2.2.1 Bunny Hunt

Bunny Hunt, the application with the smallest JavaScript codebase 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. The opposite extreme would be to download every resource on demand, as it is needed.

The Bunny Hunt code base is broken down into five files, each of which is transferred separately, as shown by an example 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. Other applications in our benchmark suite use different strategies to achieve parallel download [1]. While this may be a reasonable approach when the entire application, containing HTML and images is only 272KB, for larger benchmarks applications this is probably not the best strategy.

2.2.2 Live Maps

Live Maps is a fairly monolithic application: it loads over 900 KB of code compressed or over 3 MB uncompressed on the first page load. Figure 4 demonstrates that while much code is downloaded, only about half of it — 471 KB to be exact — is actually executed on the initial page load. While it is unlikely that the rest is dead code, it does suggest that a better splitting of code can be beneficial.

This is especially pronounced in the case of JavaScript file `ve.js`: out of over 400 KB of JavaScript, only about 100 KB is used. Furthermore, code execution takes place in "bursts", as illustrated in Figure 5: 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 100ms or so. However, many functions are not executed until 500ms later and many, as pointed out above, are not executed at all.

2.2.3 Dropthings

Dropthings is created on top of AJAX.NET, a popular AJAX toolkit and is an example of a typical framework-based application. As Figure 2 makes clear, JavaScript code constitutes over 400KB or almost 80% of the entire Dropthings transfer. Moreover, a closer examination shows that the code is not transferred very efficiently: the top-level HTML page for Dropthings declares a total of 12 files containing JavaScript, whose download is serialized. Clearly, transferring these files in parallel would have been a much better use of available client-side bandwidth. To circumvent the two-connections-per-domain limitation adopted by most Web browsers [5], several script-serving machines could be used, such as `s1.dropthings.com`, `s2.dropthings.com`, etc.

It is instructive to contrast Dropthings with Pageflakes, an industrial-strength mashup page proving the same functionality. While the download size for Pageflakes is over 1.8MB, the execution time is quite a bit faster compared to Dropthings. Examining network activity reveals that Pageflakes downloads code dynamically and only a small stub is downloadeded through a verbatim declaration as is done in Dropthings.

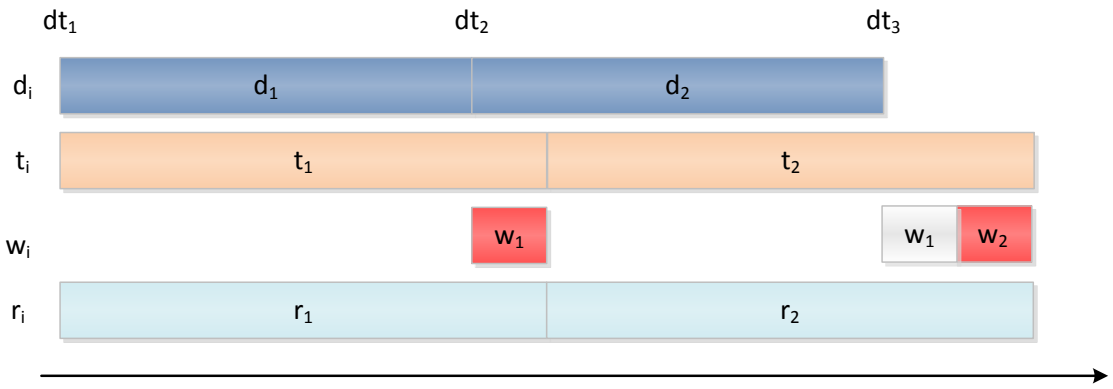

Figure 6: Two functions whose demand intervals are d_1 and d_2 and transfer costs are t_1 and t_2. Assuming that they are transferred one at a time, the waiting times are calculated as follows: $w_1 = (t_1 - d_1)^+$, $r_1 = d_1 + w_1$, $w_2 = (t_1 + t_2 - r_1 - d_2)^+$, and $r_2 = d_2 + w_2$.

3 Code Splitting Model

In our code splitting models, we consider each JavaScript file to be a collection of functions, each of which can be transferred separately. For simplicity, we ignore the issue of global variable declarations as well as top-level code, both of which are allowed by JavaScript. One way to treat such code is by introducing an artificial function `main`. Another simplification we make is that we only allow splitting at the level of top-level function declarations, treating all nested functions as an inseparable unit.

We start by collecting *demand traces* from a profile run, which reflect when each function is first executed on the client side. Collection of demand traces is performed by instrumenting function entries on the wire as JavaScript is passed from the server to the client using the AjaxScope proxy [3]. Given a demand trace, our goal is to evaluate how various code splitting strategies described below reduce the overall waiting time on the part of the user *with respect to that trace*. Clearly, for different workloads, the effect of different code splitting strategies may vary. In the future, it would be desirable to combine and average workloads from multiple users.

Definition 3.1. *Demand trace dt consists of dt_i, the first usage time for every function f_i, and dt_m, the first usage time for every code module or file m.*

Definition 3.2. *Size function s consists of s_i, the size of every top-level JavaScript function f_i, and s_m, the size of every code module or file m. Unlike demand traces, sizes are computed statically.*

3.1 Code Splitting Strategies

In this paper, we consider the following five code splitting strategies in addition to the original approach of transferring the entire file at once:

1. **Per-function**: In this scheme each function is transferred and processed *separately*. While this allows us to pipeline function transfer, this scheme also suffers from the per-file processing overhead.
2. **Greedy function**: We dynamically decide for each function whether to transfer it separately or in combination with the previous function or sequence of functions. The greedy heuristic chooses the choice that minimizes the waiting time for each function (but not necessarily for all of them). The greedy algorithm is described in Section 3.2.
3. **Per-block**: We use a heuristic to group functions that have similar demand times into a *block*. We typically end up with 3–6 blocks per JavaScrip file. The code is transferred one block at a time.
4. **Greedy blocking**: We dynamically decide for each block whether to transfer it separately or in combination with the previous block or sequence of blocks. The greedy heuristic chooses the choice that minimizes the waiting time for each block (but not necessarily for all blocks). The greedy algorithm is described in Section 3.2.
5. **Sequentially optimal blocking**: We consider all possible permutations of blocks to find the one resulting in the best overall transfer time. This is generally computationally feasible because the number of blocks per file is small, ranging from 3 to 10.

Definition 3.3. *A code splitting model M estimates the total waiting time the user experiences for a particular usage scenario w given a tuple $\langle l, b, \delta, \gamma, dt, s \rangle$, where*

- *l is the network latency;*
- *b is the network bandwidth;*
- *the per-file processing time that includes parsing within the browser is given by $\delta + \gamma \cdot s_m$;*

- *dt is the demand trace; and*
- *s is the size function.*

In our model, we make the assumption that both the latency and the bandwidth are constant over time. Moreover, as a simplification, we assume the packet drop rate to be zero; incorporating a non-zero packet drop rate could be accomplished with changing the per-file transfer time. Given the model parameters, we can easily calculate the time it takes to transfer a single function or a file. The basic transfer time for function f_i is

$$\frac{s_i}{b} + \gamma \cdot s_i = (\frac{1}{b} + \gamma) \cdot s_i.$$

If the function is transferred separately, the per-file processing cost δ is added. If the function transfer is not pipelined with other functions, the latency l is added.

3.2 Characterizing the Waiting Time

Given an execution profile and a code splitting model, we define the following for each JavaScript function:

demand interval d_i is the time after the use of f_{i-1} (or the beginning if $i = 1$) and before the use of f_i in the demand trace. It is computed from the demand trace by setting $d_1 = dt_1$ and $d_i = dt_i - dt_{i-1}$ for $i = 2, \ldots, n$.

transfer cost t_i is the time needed to transfer f_i based on the network model, which includes the cost of client-side processing. The cost differs depending on whether f_i starts a new transfer or it is included in the same (block) transfer as its predecessor.

From the demand interval and transfer cost, we compute the total waiting time by iteratively defining the following two attributes for each function

waiting time w_i is the waiting time for f_i, which is the cumulative waiting time up to the use of f_i minus the cumulative waiting time up to the use of f_{i-1}.

real use r_i is the actual use time, which equals to the demand interval plus the waiting time for f_i.

Since the quantities are defined for each function, their sums define the total execution time, $T = \sum r_i$, and the total waiting time, $W = \sum w_i$. The waiting times w_i and d_i form a partition of the execution, so

$$\sum w_i + \sum d_i = T = \sum r_i.$$

3.2.1 Per-function Splitting

Take the simple case where each function is transferred individually. The exact formulae for w_i and r_i are:

$$\begin{array}{lcll} r_0 & = & 0 & \\ w_i & = & (\sum_{j=1}^{i} t_j - \sum_{j=0}^{i-1} r_i - d_i)^+ & i = 1 \ldots n \\ r_i & = & w_i + d_i & i = 1 \ldots n \end{array} \quad (1)$$

where $(x)^+$ means to take the non-negative part of x

$$(x)^+ = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Example 1. The example in Figure 6 illustrates the basic quantities. In the figure, two functions are transferred one at a time, and the waiting time is computed for each function. The first function is delayed by the difference between the available time t_1 and the demand interval d_1. The second function becomes available at time $t_1 + t_2$. The quantity r_i is used to prevent double counting of the same waiting time by w_1 and w_2. It records the sum of the demand interval and the waiting time for each function. The quantities are computed iteratively. □

3.2.2 Per-block Splitting

When a group of functions are transferred together in one block, the wait happens only for the first function, and the transfer cost for later functions is lower than when they are shipped individually. Assuming the block begins with function f_h and ends with function f_e (for $e - h + 1$ functions in the block), the waiting times are

$$\begin{array}{l} w_h = (\sum_{j=1}^{e} t_i - \sum_{j=0}^{h-1} r_i - d_h)^+, 1 \leq h \leq n \\ w_k = 0, \quad h + 1 \leq k \leq e \end{array} \quad (2)$$

When $h = e$, the formula degenerates into the per-function case given in Equation 1. When $h = 1$ and $e = n$, it is the default case where the entire file is transferred before any function is used. The total waiting time is

$$W = w_1 = (\sum_{j=1}^{n} t_i - d_1)^+ \quad (3)$$

The cost is the positive part of the transfer time of the file minus the demand interval of the first function. If the file is transferred in multiple blocks, the formula of Equation 2 is used to compute the waiting time for each block.

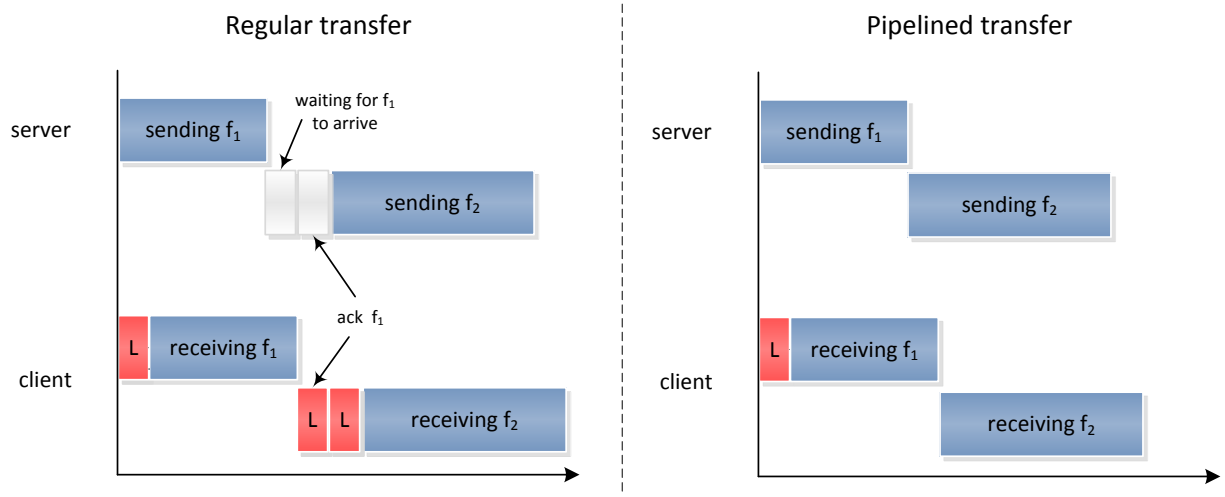

Figure 7: Differences between regular and pipelined transfer strategies.

3.2.3 Greedy Blocking

The algorithm in Figure 8 transfers the code in one or more files using a greedy heuristic. Given a demand trace, the algorithm decides, for each unit of code, whether to pack it in the same transfer with its predecessors or to split it out into a new transfer. At each step, a quantitative comparison is performed between the additional cost of packing, which is computed using Equation 2 and the cost of starting a new file.

The greedy scheme tries to minimize the waiting time for each function in demand order, and the overall result is not necessarily optimal in that it may not minimize the total waiting time, as illustrated by the example below.

Example 2. Consider a sequence of four functions f_1, f_2, f_3, f_4, whose demand intervals are $d_1 = 10$, $d_2 = 6$, $d_3 = 1$, $d_4 = 5$. Assume that the per byte cost $s_1 = 5$, $s_2 = 1$, $s_3 = 5$, $s_4 = 5$ and the per file cost δ is 5. The greedy algorithm would split at the second function because packing it leads to a waiting time of 1 but splitting it means no wait for the first two functions.

However, the best overall scheme for the four functions is to group the first three in the first file and transfer the last one in the second file. The waiting time is 7. Since the greedy algorithm splits the second function into a new file, the lowest waiting it can achieve is 9. □

$h = 1$
$r_0 = 0$
$t_1 = f(cf, size_1)$
$w_1 = (t_1 - d_1)^+$
$r_1 = w_1 + d_1$
for $i = 2$ to n
  $t_{split} = f(size_i)$
  $t_{pack} = t_{nopack} + ...$
  $w_{split} = (\sum_{j=1}^{i-1} t_j + t_{split} - \sum_{j=1}^{i-1} r_j - d_i)^+$
  $w_{pack} = (\sum_{j=1}^{i-1} + t_{pack} - \sum_{j=1}^{h-1} r_j - d_h)^+$
   $-(\sum_{j=1}^{i-1} - \sum_{j=1}^{h-1} r_j - d_h)^+$
  if $(w_{split} < w_{pack})$
    $t_i = t_{split}$
    $w_i = w_{split}$
    $r_i = w_i + d_i$
    $h = i$
  else
    $t_i = t_{pack}$
    $w_h = w_h + w_{pack}$
    $r_h = r_h + w_{pack}$
    $w_i = 0$
    $r_i = d_i$
end for

Figure 8: Greedy heuristic algorithm for selecting block splitting

3.2.4 Optimal Blocking

To find the best code splitting scheme, one can test all possible splitting choices and select the one with the lowest overall waiting time. The number of choices one needs to examine can be reduced by observing that to minimize the waiting time one should transfer the code *in order of demand times*.

To see this, consider a situation when code unit a is demanded before b but transferred in a file after the file containing code unit b. If we exchange the two functions in the two files, the second file still arrives at the same time, say t_x. Consider the use time of a and b. In the original case, the former can be no earlier than t_x but in the exchanged case, a is either used at the same or an

earlier time. Therefore, b is used at the same or an earlier time. The waiting time up to b is either the same or lower, so is the overall waiting time.

As a result, we just need to examine the choices where the code units are transferred in the order they are used, because if for any waiting time obtained by an out-of-order transfer scheme, there exists an in-order transfer scheme that has the same or lower waiting time.

Given n units of code, there are 2^{n-1} ways to divide it into different files. To see this, number the gaps between units from 1 to $n-1$. Any subset of the set $c = 1, ..., n-1$ represents one and only one type of partition of the code. For example, the empty set means to transfer all units in one file, and the complete set means to transfer the units one in each file. When n is small, we can enumerate all partitions and find the minimal waiting time, which we take as the optimal solution and will compare it with the greedy scheme.

3.3 Caveats of the Analytical Model

To make our model formulation simpler to reason about analytically, we have made several simplifying assumptions, as listed below.

- While our model considers the transfer cost to be linear in the size of the data being transferred, in reality, data is transferred as a series of network packets. While we may only use a small portion of the packet if we are, for instance, transferring a single function, we are still effectively paying for the rest of the packet to be transferred. In practice, however, the per-file penalty incurred for parsing, etc. prohibits a code splitting scheme from breaking code down into units that are too small.

- The default approach to file transfer by the browser requires that every response be acknowledged before the next request is issued by the browser. As Figure 7 illustrates, this can get pretty costly on a high-latency connection. With a world-wide average network latency of 300ms, the cost of a round-trip is 600ms. Added to the cost of in-browser processing, we have over 1 second on average to process a single file.

  An alternative is to use *HTTP pipelining*, a mechanism that allows subsequent HTTP requests to be issued without waiting for the response to arrive [2, 9]. Both our model as well as the experiments described in Section 4 assume that HTTP pipelining is implemented. For an example of how HTTP pipelining affects the waiting time for two subsequently transferred functions, consider Figure 7. The non-pipelined version spends extra $2 \times L$ for every unit of transfer compared to the pipelined version, in which the latency L is only incurred once in the beginning of pipelining.

4 Experimental Results

In this section we analyze the performance of splitting strategies outlined above with respect to the popular Web 2.0 benchmark applications described in Section 2.

4.1 Experimental Setup

To collect demand traces, we used a modified version of the AjaxScope proxy [3] which for every JavaScript block recorded the following information:

- request time;
- response time;
- size of the JavaScript block;
- size of every JavaScript function;
- number of JavaScript functions retieved.

Additionally, every declared JavaScript function was instrumented to record its start time, from which we computed the first execution time. Our experiments were performed on a Pentium 4 3.6 GHz machine equipped with 3 GB of memory running Windows XP SP2.

4.2 Experimental Data Analysis

Figure 9 shows the savings achieved by optimal blocking compared with the default transfer strategy as both absolute values in seconds and also as fractions of the original waiting time. In many cases, the savings achieved with the optimal splitting strategy are quite significant, exceeding 90% for Pageflakes with over 70Figures 10 — 15 summarize the waiting time, in seconds, for a range of network bandwidth values, starting from a 12k modem to a 1Mbit link. We consider a total of six transfer strategies: the original approach of transferring the entire file at once and the five splitting techniques summarized in Section 3.1.

Overall, splitting has the biggest advantage for narrow-band connections, ranging from a modem to slow wireless. Waiting time savings range from dozens to hundreds of seconds for monolithic applications such as Live Maps, file `ve.js`. Even for a 1Mb connection, the difference between transferring the entire file at once and optimal blocking is 6.1 seconds. For wireless-speed connections, savings cause by file splitting amount to several to several dozen seconds in the case of bigger files, which is quite significant.

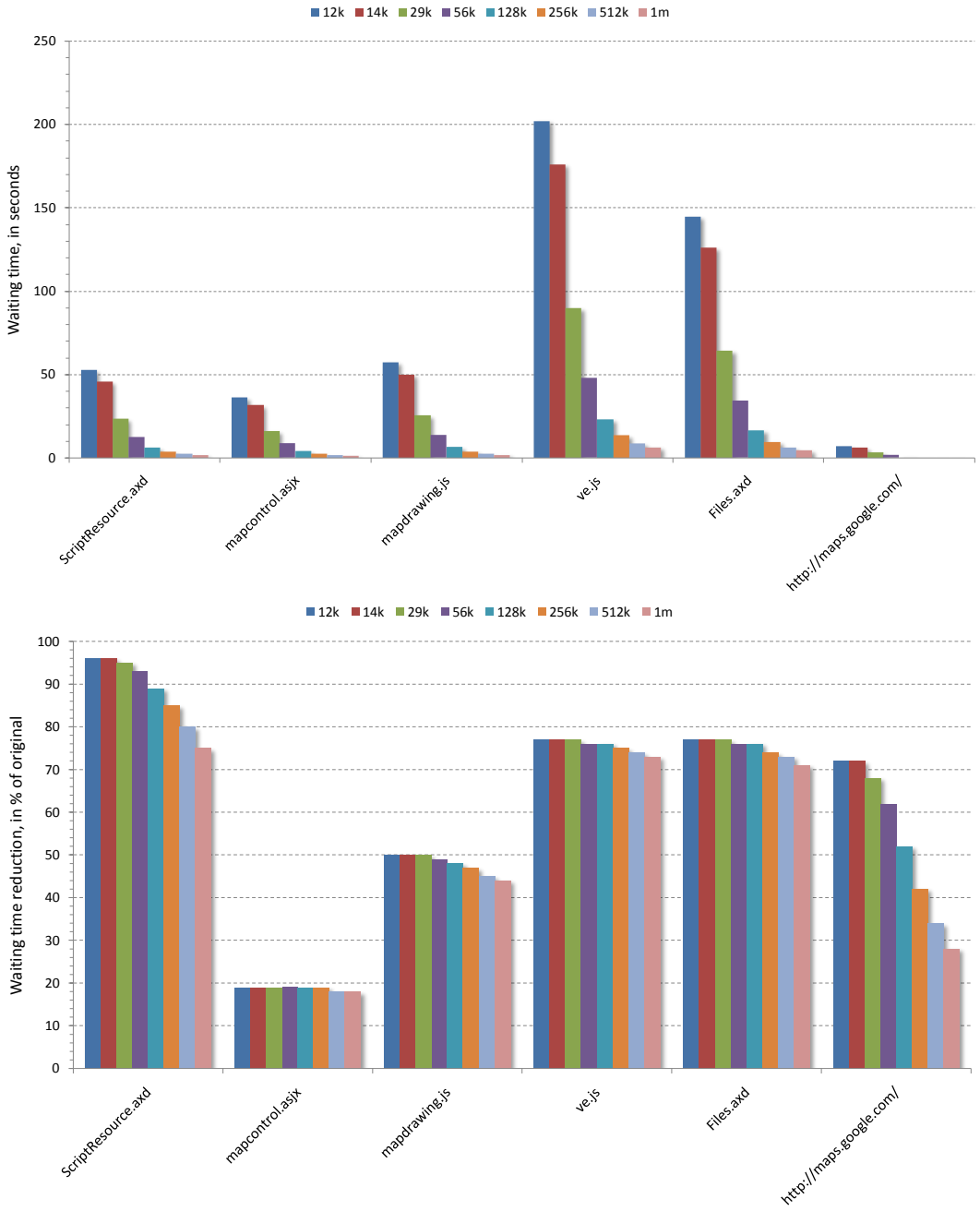

Figure 9: Comparison between the default and optimal waiting times as (a) absolute values and (b) percentages of time savings

5 Related Work

While we are not aware of research directly pertaining to responsiveness of Web 2.0 applications, several projects 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 [4]. 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 analysis can be seen as an extension of their technique, in particular, transfer blocks we identify represent "degrees of urgency": the first block 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 consti-

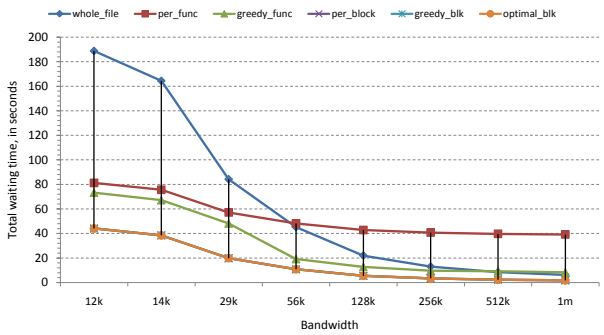

Figure 10: Pageflakes, JavaScript file `Files.axd`, size 288 KB, used 68 KB

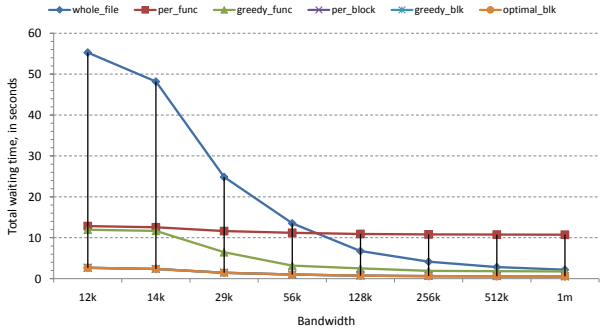

Figure 11: Dropthings, JavaScript file `ScriptResource.axd`, size 84 KB, used 7 KB and 40 top-level functions

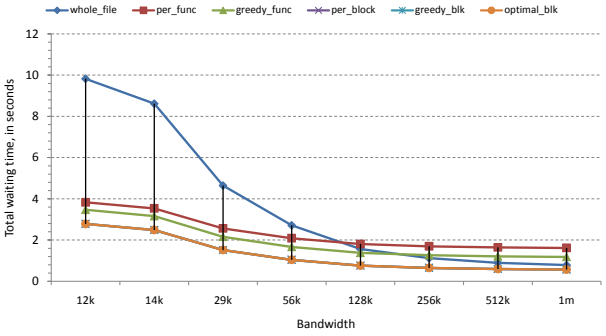

Figure 12: Google Maps, JavaScript embedded within `http://maps.google.com/`, size 27 KB, used 5 KB and 24 top-level functions

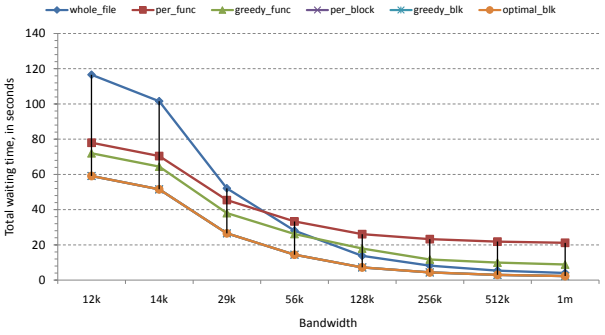

Figure 13: Live Maps, JavaScript file within `mapdrawing.js`, size 178 KB, used 111 KB and 99 top-level functions, 4 blocks

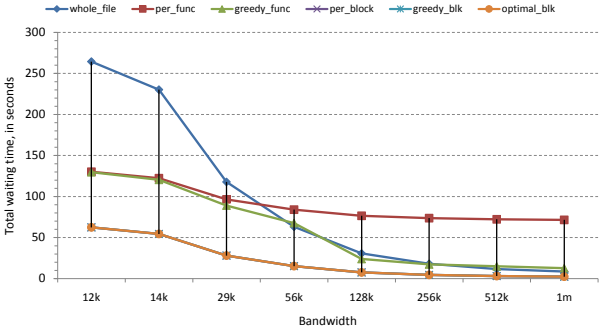

Figure 14: Live Maps, JavaScript file `ve.js`, file size 404 KB, out of that, 116 KB are used, which includes calling 215 top-level functions, 3 blocks

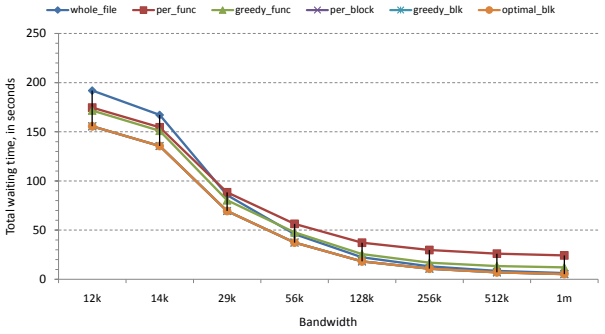

Figure 15: Live Maps, JavaScript file `mapcontrol.asjx`, size 293 KB, used 243 KB and 60 top-level functions, 4 blocks

tutes a significant portion of the application as explained in Section 2.1.

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 [7, 8]. 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 AJA.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.

6 Conclusions

In this paper we have explored code splitting as a means to improve the end-user responsiveness of large Web 2.0 applications. We have proposed five code splitting strate-

gies that apply in a wide variety of settings and evaluated their efficacy on a range of widely-used sophisticated Web 2.0 applications. To perform our evaluation, we have constructed a formal analytical model that allows us to estimate the end-user waiting times. In many cases, improvements achieved with our proposed splitting schemes were significant, ranging to dozen of seconds. Furthermore, in most cases, the savings of the waiting time saved with our best algorithm are over 70% of the original.

References

[1] K. Henriksson. Loading JavaScript files in parallel. http://blogs.msdn.com/kristoffer/archive/2006/12/22/loading-javascript-files-in-parallel.aspx, Dec. 2006.

[2] A. Hopkins. Optimizing page load time. http://www.die.net/musings/page_load_time/, 2006.

[3] E. Kiciman 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.

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

[5] E. Law. Internet Explorer and connection limits. http://blogs.msdn.com/ie/archive/2005/04/11/407189.aspx, Apr. 2005.

[6] M. Mahemoff. *Ajax Design Patterns*. O'Reilly Media, Inc., 2006.

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

[8] 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.

[9] Wikipedia. HTTP pipelining. http://en.wikipedia.org/wiki/HTTP_pipelining.