MapJAX: Data Structure Abstractions for Asynchronous Web Applications

Daniel S. Myers
MIT CSAIL

Jennifer N. Carlisle
MIT CSAIL

James A. Cowling
MIT CSAIL

Barbara H. Liskov
MIT CSAIL

Abstract

The current approach to developing rich, interactive web applications relies on asynchronous RPCs (Remote Procedure Calls) to fetch new data to be displayed by the client. We argue that for the majority of web applications, this RPC-based model is not the correct abstraction: it forces programmers to use an awkward continuation-passing style of programming and to expend too much effort manually transferring data. We propose a new programming model, MapJAX, to remedy these problems. MapJAX provides the abstraction of data structures shared between the browser and the server, based on the familiar primitives of objects, locks, and threads. MapJAX also provides additional features (parallel **for** loops and prefetching) that help developers minimize response times in their applications. MapJAX thus allows developers to focus on what they do best–writing compelling applications–rather than worrying about systems issues of data transfer and callback management.

We describe the design and implementation of the MapJAX framework and show its use in three prototypical web applications: a mapping application, an email client, and a search-autocomplete application. We evaluate the performance of these applications under realistic Internet latency and bandwidth constraints and find that the unoptimized MapJAX versions perform comparably to the standard AJAX versions, while MapJAX performance optimizations can dramatically improve performance, by close to a factor of 2 relative to non-MapJAX code in some cases.

1 Introduction

> *"It is really, really, really hard to build something like Gmail and Google Maps," said David Mendels, general manager of platform products for Macromedia. "Google hired rocket scientists... Most companies can't go and repeat what Google has done." [1]*

Recent months have shown an explosive growth in rich, interactive content on the World Wide Web — a phenomenon termed *Web 2.0*. Central to this growth is a communication technique known as AJAX: Asynchronous Javascript and XML [2]. Before AJAX, web applications were forced to fetch an entire page from the server in order to display any new data. By contrast, AJAX allows Javascript programs to send requests to the server without reloading the page or blocking the user interface. This has permitted the development of a new class of highly-responsive, desktop-like applications on the web. Moreover, support for the underlying AJAX mechanisms is ubiquitous, having been present in web browsers since the late 1990s, so these applications can be delivered without the need for third-party plugins.

The current AJAX programming model has two significant shortcomings, however. First, AJAX requires that web clients request content from web servers using asynchronous HTTP requests: the client bundles any statements that depend on the result of the request into a callback that will be executed when the response to the HTTP request arrives. This approach forces programmers to use an awkward continuation-passing programming style and thread program state through a series of callback functions. While various toolkits [9, 15] elevate the level of abstraction to that of an RPC, none has eliminated the use of continuations and callbacks.

Additionally, a programmer using AJAX must develop his or her own techniques for avoiding delays associated with fetching content from the server. These delays can be reduced by prefetching content before it is needed, and by sending requests in parallel. Neither AJAX nor current tools built on top of it provides direct support for these approaches.

This paper presents MapJAX, a data-centric framework for building AJAX applications without any new client-side software. In place of asynchronous RPCs, MapJAX provides the programmer with the illusion of logical data structures shared between the client and server. These data structures appear to be ordinary objects that can be accessed through normal Javascript method calls, hiding the underlying complexity of continuations and callback functions. Instead, the code that causes a fetch of data from the server can be thought of as running in its own thread. The thread blocks while the call is being processed and then continues running when the server response arrives. The MapJAX approach thus allows users to create programs with the mental model to

which they have grown accustomed: threads and objects.

In addition, MapJAX also provides a number of mechanisms that enable efficient interaction between client and server. Given that the cost of communication far exceeds the cost of computation in this setting, we focus on ways to reduce communication delays. First, MapJAX allows application programmers to decrease the latency involved in data structure access by using spare bandwidth to prefetch objects. The MapJAX runtime maintains a cache of previously fetched objects and avoids communication delay when the requested content is present in the cache.

Second, MapJAX provides a mechanism that allows a number of fetches to be sent in parallel. This is provided in the form of a parallel **for** statement. A common use case for web applications is to copy a range of elements to the screen. This is naturally expressed as a sequential **for** loop over a shared data structure, albeit with poor performance. Instead, the parallel **for** starts up the iterations in parallel, allowing the requests to be issued immediately and concurrently.

In order to allow applications to express ordering constraints on these loops while preserving concurrency, MapJAX provides a new locking mechanism that allows a thread to reserve a lock in advance of acquiring it. Lock reservation is non-blocking and simply places the identifier of the thread into the lock queue. Later, the thread acquires the lock using an acquisition function which blocks until the lock is available. As detailed in Section 3.5, this model allows threads to generate requests in parallel, yet process their responses in order.

MapJAX also provides a few additional features to increase the effectiveness of prefetching and parallel **for**. It is able to group a number of requests into one communication with the server, and it allows the program to cancel requests that are no longer needed.

MapJAX is defined as a small veneer over Javascript, as a goal of the system is to require a minimum of programmer retraining. Its implementation consists of a compiler that translates MapJAX programs to Javascript, and a runtime system written in Javascript that provides the needed support for the MapJAX features. Despite significant work on programming environments for web applications [13, 10, 9, 3, 15, 4, 11], we are not aware of any existing work that provides a programming model like that of MapJAX.

We have used MapJAX to implement three prototypical web applications–a mapping application, an email client, and a search autocomplete application. These were all implemented with relative ease, while benefiting from prefetching and parallel **for** loops. Our results show that MapJAX has minimal overhead compared to an equivalent AJAX program. The results also show that use of our advanced features resulted in substantial performance improvements. to as much as a factor of 2 in some cases.

2 Javascript and AJAX Applications

We first provide a brief overview of Javascript and AJAX. The Javascript language was originally developed at Netscape in 1996 to allow interactivity in web pages. In particular, Javascript allows handlers to be registered for activation in response to various events that can occur on the page, such as the page being loaded, the user clicking on a link, moving his or her mouse over an image, and so forth. While Javascript has seen use outside of the context of web pages, within this context Javascript programs are event-based. As only one event handler can execute concurrently and scheduling is non-preemptive, Javascript programs can be viewed as executing under a single-process, event driven (SPED) model similar to e.g. Zeus [17].

Application programmers can modify web page content from Javascript using the Document Object Model (DOM), a programmatically-accessible, tree-structured representation of the elements on the page.

Until relatively recently, Javascript was used only for purely client-side tasks, such as verifying text input into a zip-code form field. Since the late 1990's however, Javascript has contained an "XMLHttpRequest" object, allowing programmers to send asynchronously-handled HTTP requests for XML-formatted data. The realization in the web development community that this object could be used to fetch new data without reloading a page gave rise to AJAX, standing for "Asynchronous Javascript and XML", although more recently alternative encodings such as Javascript Object Notation (JSON) [5] have been used in place of XML.

3 Programming Model

This section describes the MapJAX programming model. We begin with the basic features that allow programmers to write working programs (shared data structures and non-preemptive threads), then describe various features that help them to improve performance (data prefetching, parallel **for** loops, RLU locks, and request canceling).

3.1 MapJAX Maps

MapJAX objects represent logical data structures shared between the client and server, and are at the core of the MapJAX system. These objects are collections of elements, e.g., a collection of email messages, or a collection of grid cells in a mapping application. Each map is associated with a URL at the server to which requests are sent to retrieve elements.

- *Cmap(String url, Object prefetch, String transport)*. Creates a new MapJAX map, where *url* is the URL that will be used to fetch its elements, *prefetch* is the prefetching policy, and *transport* identifies the type of transport used to make requests.
- *Object access(String key)*. Returns the element named by *key*. Alternatively, returns a failure code if the *key* is invalid or the network fails.
- *Boolean has(String key)*. Returns true if the element named by *key* is present in the cache.
- *void prefetch(String[] keys)*. User-initiated prefetching.
- *void cancel(String key)*. Cancel an outstanding access.

Figure 1: MapJAX maps API.

The base representation provided by MapJAX for collections of elements is a map from keys to values, where the keys are strings, and the values can be any Javascript data type (we assume that maps do not contain other maps). Maps are supported at the server by this base representation. At the client side, however, it can be useful to access the shared structure at a higher level of abstraction, such as an array or tree.

MapJAX provides three higher-level abstractions: one and two-dimensional arrays, and trees, with the class names ARRAYMAP, GRIDMAP, and TREEMAP, respectively. One dimensional arrays use integers as keys, two dimensional arrays use pairs of integers as keys, and trees use strings where each string represents the path from the root to the element of interest. Programmers are free to implement their own abstractions using any of these primitives, and MapJAX is capable of supporting general object graphs.

MapJAX objects appear as ordinary Javascript objects. Figure 1 shows the interface to MapJAX maps; although Javascript is not statically typed, we have included types in the method descriptions to clarify the presentation.

The constructor takes as an argument the URL to be used to fetch elements of the map. It also takes a *prefetch* object, which defines how prefetching works for this map; prefetching is described in Section 3.3. The third argument specifies the type of transport to be used to retrieve elements; we defer discussion of this implementation detail to Section 4.

The most interesting method of a MapJAX object is *access*. A call of this method, e.g., *foo.access("bar")*, is a blocking call: it will not return until the requested element has been retrieved from the server, although it may return immediately if the value is already cached. The MapJAX threading model (discussed below) allows other Javascript or browser code to execute while the call is blocked.

The *has*, *prefetch*, and *cancel* methods are used for cache management and server functionality; they are discussed in later sections.

Maps are read-only. Given the generally read-oriented nature of the web, we do not feel this to be an overly onerous limitation, but leave write-enabled structures as an area of future work.

3.2 Threads

A MapJAX program consists of a collection of *threads*. A new thread is created each time an event handler is invoked by the browser. Threads are also created for iterations in the parallel **for** as discussed in Section 3.4.

Threads are scheduled non-preemptively: a thread retains control until relinquishing it. A thread relinquishes control when it finishes processing an event or when it makes a blocking call on a MapJAX object. For example, when a thread calls the *access* method of a MapJAX map object, causing an RPC to be made to the server, it relinquishes control; it will regain control at some point after the result of the RPC arrives (or the RPC is aborted because of communication problems).

Threads are implicit at present, and relinquishing control is also implicit. However, it would not be difficult to extend MapJAX to support explicit threads and thread management (e.g., a **yield** statement) if this turned out to be useful.

It is worth pointing out that the concurrency in MapJAX programs also exists in AJAX programs. The difference is that in MapJAX programmers can think of each event handler as running in its own thread, with the system switching control among threads automatically, whereas in AJAX, the programmer needs to write callbacks and continuation functions.

3.3 Prefetching

All MapJAX maps support prefetching via programmer-defined prefetching policies. A prefetching policy is a Javascript object. It provides a method, *getPrefetchSet*, that, given a key, returns a set of keys that identify elements to prefetch. Prefetching policies are usually specialized to the kind of MapJAX map in use in the web application. E.g., for a map-viewing application, the prefetching policy might indicate to fetch all grid cells adjacent to the one being requested. Additionally, the prefetching policy can be tailored to a particular higher level abstraction; e.g., one defined for arrays would expect keys to be integers.

Figure 2 provides an example prefetching object that implements a "read-ahead K" policy for an ARRAYMAP.

```
// Javascript object constructor
function ReadAheadKPolicy(k) {
    this.k = k;
}
// Javascript object definition
ReadAheadPolicy.prototype = {
    getPrefetchSet: function(idx) {
        var pf_set = new Array();
        for (var i = 1; i <= k; ++i) {
            pf_set.push(idx + k);
        }
        return pf_set;
    }
}
```

Figure 2: Example read-ahead-k prefetching policy for an ARRAYMAP.

The *getPrefetchSet* method merely identifies elements of interest. The MapJAX runtime ensures that elements already present in the cache will not be refetched, so these policies do not need to be aware of the state of the cache.

Calls to *getPrefetchSet* are made automatically as part of processing a call to the *access* method. *Access* calls *getPrefetchSet* on the prefetch policy object associated with the map and then requests a fetch of the original key plus all the keys returned by the call. A programmer can also initiate ad-hoc prefetching by calling the *prefetch* method of a map object with an array of keys to prefetch. This method informs the MapJAX runtime of the need to prefetch the elements and then returns immediately (it is non-blocking). The actual fetching occurs in the background.

Custom prefetch policies can be written with a minimal amount of effort from application programmers, allowing prefetching to be tailored to specific web applications. We note additionally that these policies need not be static: as full-fledged Javascript objects, they can maintain internal state to adapt based on the request history.

3.4 Parallel for Loops

A common use case for web applications is to copy a range of elements to the display. The obvious way to program this is to use a **for** loop, where each iteration is responsible for fetching and rendering each element. A sequential execution model is not well suited for the processing of such a loop, however, since it will force one iteration to complete before the next iteration begins; in particular, this will needlessly delay the launch of RPCs to fetch missing data. Prefetching helps but does not completely solve the problem.

To optimize this common and important case, we introduce a parallel **for** statement into the MapJAX language, written **pfor**. The semantics of this statement are as follows. Each iteration runs in a separate thread. Control starts with the first loop iteration; as soon as it blocks, the next iteration starts to run, and so on. More formally, we guarantee that each iteration will initially be given a chance to run in loop order; after the thread corresponding to that iteration yields control, however, it regains control in an arbitrary order with respect to other threads in the loop. Locks, described in the next section, can be used to impose additional ordering constraints if need be. Control passes to code following the loop only once all the iteration threads have terminated.

Our parallel **for** loops are thus similar to standard parallel **for** loops in that they require loop iterations not to effect the termination condition of the loop. They differ slightly, however, because they explicitly start iterations in loop order. Combined with our novel locks, discussed below, this allows programmers to enforce useful ordering constraints that could not be captured with a standard parallel **for**.

The use of the parallel **for** statement can provide considerable performance benefits, as discussed further in Section 6.

3.5 Locks

Any language with concurrency requires some mechanism for its control. In MapJAX, we provide programmers with a novel type of local lock. These locks can be used in the normal way: first a thread acquires the lock and later releases it. However, our locks also provide the ability to reserve the lock in advance of acquiring it. We call these RLU locks because of the "reserve/lock/unlock" regime for using them.

Reserving a lock doesn't block the thread; instead it records the thread on the end of a reservation list. When a thread executes the *lock* method, it will be delayed until the lock is available *and* it is the earliest thread on the list. The interface for MapJAX locks is given in Figure 3. Note that a thread can call the *lock* method without having previously reserved the lock. In addition, the *unreserve* method can be called to give up a reservation.

RLU locks are motivated in large part by our **pfor** loops. Consider a **pfor** loop in which the programmer intends for each iteration to update a shared variable in loop order using data fetched from the web server through a shared map. While the iterations are started in loop order, the responses to the fetch requests may arrive in a different order.

With normal locks, the only solution would be for each iteration to acquire the lock on the shared variable *before* performing the *access* call, thus preventing the iterations from making their fetch requests in parallel. With RLU locks, threads reserve the lock in loop order. They may

- *RLUlock( )*. Creates a new lock object. The lock is available and the reservation list is empty.
- *void reserve( )*. If the thread is already on the reservation list for the lock object, does nothing. Otherwise adds the thread to the end of the reservation list.
- *void lock( )*. If the thread isn't on the reservation list, adds it to the end of the list. Blocks until this thread is at the front of the list and the lock is available. Then acquires the lock and removes the thread from the list.
- *void unlock( )*. If this thread holds the lock, releases the lock, else does nothing.
- *void unreserve( )*. If this thread is on the reservation list, removes it from the list, else does nothing.

Figure 3: MapJAX RLU locks API.

then immediately call *access* and initiate the transfer of remote data, blocking to acquire the lock only when absolutely necessary (before updating the shared variable). The code is given in Figure 4.

```
var sharedData = new ArrayMap(...);
var l = new RLULock();
pfor(var i = 0; i < 47; ++i) {
  l.reserve( );
  var newData = sharedData.access(i);
  l.lock( );
  localObject += newData;
  l.unlock( );
}
```

Figure 4: Locking example.

Finally, we note that RLU locks are useful outside of a **pfor** statement as well: the network reordering issues they are designed to address can arise any time multiple threads seek to synchronize their accesses to an object, and they can also be used as normal locks.

3.6 Request Canceling

When bandwidth is limited, applications must manage it carefully. Request canceling is one mechanism by which they may do so. Specifically, sometimes an application can determine that certain data elements are no longer needed. For example, in our mapping application, a user scrolling quickly in a low-bandwidth environment can trigger two updates for the same cell on screen, where only the latter update is required. If the application can indicate to the runtime that the first update is no longer needed, considerable bandwidth can be saved in the case where the RPC for the first request has not yet been sent to the server.

Request canceling is supported by the *cancel* method of MapJAX maps. This call takes a key, *k*, as an argument. If there is no outstanding RPC with *k* as an argument, the cancel request has no effect. Otherwise, the RPC with *k* as an argument *might* be canceled; other RPCs might be canceled as well. If an RPC is canceled, any call of *access* that is waiting for the results of that RPC will return with a failure code. The failure code allows the thread that is waiting for the result to act appropriately when it starts running again.

The MapJAX runtime determines what is canceled based on heuristics about the utility of the outstanding RPCs. Generally, if *k* was an argument to a previous *access* call, the system will cancel all RPCs corresponding to that call, i.e., requests both for *k* and for other keys identified by prefetching. However it will not cancel an RPC for one of the prefetch keys if it appeared as an argument to a later *access* call or contains requests for non-canceled items. More details about how canceling works can be found in Section 4.2.4. Note that programmers needn't be concerned with these details; instead they simply cancel based on knowledge of what is no longer needed, and the runtime makes the ultimate decision.

4 Implementation

This section describes the implementation of MapJAX. MapJAX is presented to the user as a small extension to the standard Javascript language. The MapJAX implementation has three parts: a compiler that translates MapJAX programs to standard Javascript, a client-side runtime Javascript library that implements the majority of the MapJAX programming model, and a server-side library.

The current version of MapJAX implements the vast majority of the programming model described in Section 3, although it is still an unoptimized prototype. The only features not fully implemented are some request-canceling corner cases described in Section 4.2.4.

4.1 MapJAX Language and Compiler

A major goal for MapJAX is to allow programmers to access data at servers using normal method calls, thus avoiding the complexity of programming with continuations and callbacks. MapJAX also provides support for writing high-performance code, including the parallel **for** statement.

Both blocking calls and the parallel **for** statement require the use of a compiler whose job it is to produce the corresponding Javascript program. This code is based

on callbacks and continuation functions, which permit an efficient implementation.

The MapJAX com piler needs to be able to recognize the features requiring translation. We accomplish this as follows. Blocking method calls are indicated by using special names for these methods: these names always end in "#" (e.g., access#, lock#). The parallel **for** statement is indicated by using an additional keyword **pfor**. Thus our extensions to Javascript are very small. We opted for this approach because we wanted MapJAX to remain similar to Javascript, so that programmers who already knew Javascript would be able to use MapJAX without much effort.

When the compiler encounters one of the blocking method calls, it computes the continuation of the call, packages that code as a continuation function, and adds that function as an extra argument to the call. Additionally, the compiler applies this procedure transitively: any function that calls a blocking method is tagged, and the compiler applies this procedure for any call to a tagged function. Thus, code using the MapJAX programming model is converted into callback-based code compatible with standard Javascript.

The code produced for the **pfor** statement also makes use of continuations and callbacks. Here the compiler must produce code to spawn each iteration as a new thread, to produce the next thread when the previous one blocks, and to ensure that the code after the loop isn't executed until all iterations have terminated.

4.1.1 Function Denesting

Javascript supports nested function declarations, and the initial version of the compiler used them in the generated code: continuation functions were nested in the function from which they were generated, which allowed easy access to variables declared therein. (This would also be an attractive way to write standard AJAX code.) Due to the Firefox implementation of Javascript, however, performance of this code was quite poor: we found that access to variables declared in a nesting hierarchy was considerably slower than access to variables declared in a top-level function. Therefore, the compiler now performs an optimization pass in which the nested code generated by the compiler is fully de-nested; variables needed by formerly-nested functions are stored and passed explicitly in "closure objects." Nested code that existed in the original input file is not denested.

4.2 Client-side Runtime

The majority of MapJAX is implemented in the client-side runtime. Specifically, the runtime provides support for handling accesses to MapJAX objects (including RPC transmission and cache management), creating and scheduling threads of control, and locks.

4.2.1 Object Cache

The MapJAX runtime makes use of a Javascript object that implements a cache for MapJAX object elements. The cache holds previously fetched object elements. Each time a new element arrives from the server, it is added to the cache. Elements are removed from the cache based on TTLs; these TTLs are provided by the server and are sent in the RPC replies. TTLs are intended to ensure data freshness, not to manage the size of the cache. In general, we believe that cache management policies are relatively uninteresting for these applications, given the large size of the cache relative to the amount of data that would normally be downloaded in a reasonable timeframe. Adding sophisticated management policies would be straightforward and able to draw on the large body of existing work.

4.2.2 Accessing Objects

When the *access* method of a MapJAX map is invoked, it first computes the set of prefetch keys. Then it passes the requested key, the set of prefetch keys, and associated callback function (generated by the compiler) to the MapJAX runtime. The runtime interacts with the cache to determine which of the requested elements need to be fetched, and it prunes the list to remove all elements currently present in the cache.

If any elements remain after this pruning, the runtime initiates a request or requests to the server for the missing elements. Then, if the cache contains the requested element, the callback function is invoked immediately. Otherwise, the MapJAX runtime stores the callback function. When an element arrives from the server, callback functions pending on that element are invoked in the order they were submitted.

The MapJAX runtime maintains information about each pending *access* method call: it records the key requested as an argument of that call, plus any additional prefetch keys, the callback, and the RPCs generated to satisfy the request.

4.2.3 Transport Abstraction

The communication protocol is abstracted into a separate class, allowing different transports to be used to fetch objects. For example, AJAX requests through the XMLHttpRequest object can only be used to fetch text data. Binary data, such as images, are not supported. However, by substituting a class that uses the browser's support for loading images directly, rather than AJAX calls, we can support image requests using the same model as

is used for text data. The purpose of the third argument of MapJAX object constructors, left unspecified earlier, is to specify which transport should be used to service misses for that object.

4.2.4 Request Canceling

Request canceling is currently implemented in a simple way; requests are canceled if they contain a single cancelable element. A full implementation would be designed as follows. The MapJAX runtime looks through its list of outstanding *access* requests and their associated RPCs. If the key k being canceled is the key requested by some *access* request, we cancel all RPCs associated with that request except for RPCs requesting keys that are listed in more recent *access* requests. If k is not a requested key (i.e., it is a key generated by a prefetching policy), we cancel the RPC containing the request for it, provided any other keys in the request have already been canceled (the runtime carries out the necessary bookkeeping to determine if they have been).

4.2.5 Request Combining

For performance and scalability reasons, requests to the server for MapJAX object values should be grouped into single messages when possible. The MapJAX runtime cannot predict the future and doesn't know when it receives a request whether it should wait for another. However, when executing the code corresponding to a **pfor** loop, it is clearly advantageous to wait. In this case, MapJAX defers sending any requests until each iteration of the loop has been given a chance to run, offering opportunities for combining.

Requests to the server should not be allowed to grow excessively large, however, since the application must wait for an entire message to be received before beginning any processing. While MapJAX does not currently enforce a limit on message size, this functionality could be added later in a manner completely transparent to existing application code.

4.2.6 Callstack Depth Monitoring

Because the client runtime uses continuations, it is possible for the call stack to grow excessively deep. In particular, consider a (non-parallel) **for** loop over an array where all elements are already locally cached. In this case, each loop iteration will add another stack frame, and moderate-sized loops were found to exceed the maximum Javascript stack depth in practice. To cope with this issue, the compiler inserts code to track the depth of the stack at runtime, and the MapJAX runtime monitors this value. If the stack depth grows beyond a given value, the runtime will break the call chain by using Javascript-provided facilities to schedule future event execution (*window.setTimeout*) to execute the next call, rather than allowing it proceed directly.

4.3 Server-side Library

Implementing MapJAX objects requires cooperation from the server. Specifically, each object is associated with a URL on the server that accepts requests for one or more elements in the corresponding shared data structure and returns the corresponding values and TTLs. MapJAX provides a library for use in Java Servlets and Java Server pages for building such servers, but nothing about MapJAX requires the use of Java on the server-side: any software able to process HTTP requests with request parameters will suffice.

5 Applications

We have implemented a number of web applications to evaluate MapJAX based on both programming efficiency and performance. Our chosen applications replicate three prototypically successful AJAX applications: Google Suggest, Google Maps, and Gmail. Each was implemented from scratch using both standard AJAX techniques and MapJAX. Here, we describe the applications and their implementations; in Section 6, we describe their performance under the MapJAX framework.

5.1 Auto-Complete Application

The search auto-complete application, echoing the functionality of Google Suggest, is representative of applications where very low-latency fetching of server data is required. Here, a user types a search phrase into a text box within a web page, and suggested text completions are offered in real-time. A TREEMAP MapJAX object is used to implement a trie providing access to the suggestion set for each successive keypress. Cache misses must be handled with low overhead to ensure responsiveness for typists of even moderate speed.

As each keypress generates a new completion set that obsoletes any previous one, and given that the network may reorder messages, care must be taken to ensure that the correct data are always displayed. MapJAX locks provide a simple mechanism to enforce this constraint: the handling code for a keypress event simply reserves a lock on the completion display object, accesses the shared trie, then locks and updates the display object.

Given the speed at which users type, completion set prefetching can yield a noticeable improvement in application performance. Figure 5 illustrates the expression of a custom prefetching policy in the MapJAX framework.

```
EnglishPrefetchPolicy.prototype = {
  getPrefetchSet: function(idx) {
     var lastchar = idx.charAt(idx.length - 1);
     var pset = new Array();
     if (this.isVowel(lastchar))
       for (var i=0; i<this.con.length; i++) {
         pset[i] = idx + this.con[i];
       }
     } else {
       for (var i=0; i<this.vowels.length; i++) {
         pset[i] = idx + this.vowels[i];
       }
     }
     return pset;
  }
};
```

Figure 5: Implementation of a custom prefetching policy in MapJAX. This policy uses a basic model of the English language to predict future search queries based on the current query.

The prediction of the next character that the user will input is based on the last character he or she has entered: we predict that a consonant will follow a vowel, and vice-versa, which is an approximation to English word structure.

5.2 Mapping Application

Our mapping application closely resembles Google Maps: it provides the user with a mobile viewport over a large map. By clicking and dragging the viewport, the user can examine different portions of the map. This example exhibits the implementation of a complex web application (one widely claimed to be difficult to implement) with relatively little effort under the MapJAX framework. The grid nature of the data is well matched by a two-dimensional array abstraction, provided by the MapJAX GRIDMAP. While the logical dimensions of this grid are extremely large (100,000+), MapJAX requires only enough memory on the client to store the data accessed. Moreover, MapJAX is able to stream data as they are needed, rather than requiring the entire working set to be transferred at once.

The map application takes advantage of parallel data fetches from the server and uses locks to ensure that the proper elements are displayed. This application presents a particular challenge, however, because of the bandwidth requirements involved. If prefetching is implemented poorly, it can increase response times by delaying requests that satisfy cache misses. Even if prefetching decreases response times, overly-aggressive prefetching wastes bandwidth, causing the application provider to incur unnecessary bandwidth costs. In our experiments we implement a simple omnidirectional prefetch policy, OMNI, which fetches all tiles within a square of size k of the accessed tile.

When the user moves to another region of the grid, the current set of tiles will be rendered obsolete as new information is requested. Often, if the user is moving quickly through the space, tiles that have been requested for the current display have not yet arrived. Furthermore, these tiles have associated prefetch sets that have also been rendered obsolete. Fetching these unwanted tiles to satisfy outstanding access requests wastes bandwidth. With request cancellation, these requests can be eliminated.

The mapping application is not actually an AJAX application due to the inability of AJAX to transfer binary data. Both the MapJAX and non-MapJAX implementations use native browser support for loading and caching images. The example illustrates the ability of MapJAX to provide a uniform interface to data, regardless of its type.

5.3 Webmail Application

We implemented a two-pane webmail application. The left pane displays a list of email message headers (sender, date, and subject); when one of the headers is clicked, the corresponding message body is displayed in the right-hand pane. The left pane contains at most 40 message headers; additional message headers can be viewed by clicking a "next page" link, which loads the next 40 headers.

The usage patterns characteristic of webmail applications provide an ideal setting for MapJAX-based data prefetching. Users desire low latency when loading new screens of message headers or viewing message bodies, but they generally have long "think times" while reading messages, providing a long interval during which the system can prefetch data. MapJAX permits an implementation of this application with little programmer effort. The application programmer simply accesses headers and bodies directly from MapJAX objects, without considering data transfer explicitly except for choosing a prefetching policy.

This application also illustrates the utility of MapJAX parallel **for** loops and locks. When loading a new screen of message headers, the programmer needs to append new header objects to the header list in the appropriate order. A non-MapJAX implementation needs to implement this ordering manually, which complicates program development by forcing the programmer to manually group object requests into messages and ensure that responses are processed in order.

By contrast, the MapJAX version of the code using a parallel **for** loop is correct, simple, and fast. The code in Figure 6 loads the first 40 headers into the header list in order, regardless of how many RPCs the MapJAX runtime chooses to issue to retrieve the headers. We assume that the header list is represented by a <DIV> element

```
var mailHeaders = new ArrayMap("mailHeaders",
                      new ReadAheadKPolicy(10),
                      "AJAXTransport");
var headerList =
   document.getElementById("header_list_div");
var hdrLock = new RLULock();
pfor(var i = 0; i < 40; ++i) {
   hdrLock.reserve();
   var header = mailHeaders.access#(i);
   // Omit code to check for error condition
   hdrLock.lock#();
   var hdrDiv = document.createElement("div");
   // Omit code to initialize hdrDiv from header
   headerList.appendChild(headerDiv);
   hdrLock.unlock();
}
```

Figure 6: MapJAX implementation for loading a page of email headers.

whose ID is "header_list_div". We show the corresponding Javascript code generated by the MapJAX compiler in Figure 7.

Figure 8 shows a version of the AJAX code which retrieves all headers using a single RPC. Already, this code can be seen to be more complicated than its MapJAX counterpart; adding facilities for tracking and ordering multiple requests (not to mention calculating the appropriate number of requests to issue) would only make the situation worse.

5.4 Non-MapJAX Implementations

We close with a word on our non-MapJAX implementations of these applications. While the MapJAX applications use the automatic prefetching and caching features of the framework, we do not implement manual caching or prefetching in the non-MapJAX applications. Our rationale is that a manually-tuned application should always be able to perform as well as MapJAX, given a sufficient time investment: MapJAX and human programmers both have the same set of primitives available to them. We thus show the improvement possible given a programmer unwilling or unable to invest extensive time and energy in optimization.

Note that the absence of prefetching in the non-MapJAX examples greatly reduces their implementation complexity. Had we included this functionality, the benefits of of MapJAX would have been even more apparent. Caching on its own, by contrast, would be of little use on the test workloads presented in Section 6, as they do not reuse data, and the non-MapJAX applications do not suffer from its absence.

Caching and prefetching aside, we have attempted to write implementations that, while straightforward, avoid obvious performance pitfalls. We describe each in more detail below.

The non-MapJAX version of the webmail application

```
var mailHeaders = new ArrayMap("mailHeaders",
                      new ReadAheadKPolicy(10),
                      "AJAXTransport");
var headerList =
   document.getElementById("header_list_div");
var hdrLock = new RLULock();
for(var i = 0; i < 40; ++i) {
  hdrLock.reserve();
  _cx_thread_create(function() {
    mailHeaders.access(i, _cx_cont1);
  });
}

function _cx_cont1(header) {
    var contobj = new Object();
    contobj.header = header;
    hdrLock.lock(_cx_cont2, contobj);
}

function _cx_cont2(contobj) {
    var header = contobj.header;
    var hdrDiv = document.createElement(''div'');
    // Omit code to initialize hdrDiv from header
    headerList.appendChild(headerDiv);
    hdrLock.unlock();
}
```

Figure 7: Javascript code produced by the MapJAX compiler for loading a page of email headers.

takes the one-RPC approach to fetching message headers as discussed in Section 5.3 and illustrated in Figure 8. Message bodies are retrieved using one RPC per body, as multiple bodies are never fetched simultaneously.

The non-MapJAX version of the suggest application is almost identical to the MapJAX version, except that it explicitly sends an RPC to fetch completion requests. In order to cope with network reordering, it maintains a version number on the completion display field that it increments each time it sends an RPC. The callback function for each RPC has a copy of the version number with which it is associated; when it is run, it checks the version number on the completion display and only updates the display if the version matches. For compatibility with the Google version of the application, whose data we use, RPC results contain Javascript code which is `eval'd` to update the display.

The non-MapJAX version of the mapping application is also close in implementation to the MapJAX version. Using much the same event-handler code, it moves and updates a collection of image objects on screen, the primary difference being that new image data are loaded by setting the "src" attribute of these objects to the appropriate URL, rather than using a MapJAX grid-map. While this implementation does not prefetch, the browser will cache images, and it also benefits from the request-canceling functionality that browser image objects support.

```
var req = new XMLHttpRequest();
req.open("GET", "/mail-headers.cgi?idxs=0-40");
req.onreadystatechange = headerRPCHandler;
req.send(null);

function headerRPCHandler(req) {
  if (req.readyState == 4 && req.status == 200) {
    var jsonEncodedHeaders = req.responseText;
    var headers = decodeJSON(jsonEncodedHeaders);
    var headerList =
      document.getElementById("header_list_div");
    for (var i = 0; i < headers.length; i++) {
      var hdrDiv = document.createElement("div");
      //Skip code to init hdrDiv from headers[i]
      headerList.appendChild(hdrDiv);
    }
  } else {
    if (req.readyState == 4) {
      // handle RPC error condition
    }
  }
}
```

Figure 8: Non-MapJAX, single-RPC implementation for loading a page of email headers.

6 Experimental Results

In this section, we provide performance results that demonstrate the advantages of MapJAX. We test the applications described above under realistic Internet latency and bandwidth constraints and show that the unoptimized MapJAX versions perform comparably to the standard AJAX versions, while MapJAX performance optimizations can dramatically improve performance, by up to a factor of 2 in some cases. Additionally, we provide microbenchmarks demonstrating the utility of specific features of MapJAX.

These results illustrate two points. First, they show that the more intuitive programming model of MapJAX is provided with little overhead. Second, the results demonstrate that prefetching and other MapJAX optimizations are useful tools for these kinds of applications, and that substantial performance increases can be achieved with relatively simple-minded prefetching policies. Were one willing to expend additional effort devising more clever prefetching policies or tweaking other portions of the application, one might well achieve better performance than seen here. We do not consider that fact to detract from our results.

Our experiments were conducted using two PCs with Intel Pentium 4 3.8GHz CPUs and 4GB of RAM, running Fedora Core 4 with Linux kernel version 2.6.14-1.1656.FC4_smp. The server ran Apache Tomcat 5.5.17 with the tcnative extensions, and the client web browser was Firefox 2.0.0.1. To allow effective prefetching of images, and as recommended for good AJAX performance, we modified the Firefox configuration variables as indicated in Table 1, although we note that these changes are not mandatory. Before executing each experiment, we executed and discarded a complete run to warm up the server's cache.

To introduce network delays and bandwidth constraints, we used a 600 MHz Intel Pentium III running FreeBSD 4.11 and the dummynet [12] network emulator. This machine was connected to both the client and server by 100Mbit switched Ethernet.

Parameter	Value
network.http.pipelining	true
network.http.pipelining.maxrequests	16
network.http.max-connections	48
network.http.max-connections-per-server	48
network.http.max-persistent-c'xns-per-server	8

Table 1: Firefox configuration variables changed from defaults during testing.

6.1 Application Tests

6.1.1 Search Auto-Complete

As stated earlier, the search term auto-complete application is interesting because it represents an application with low bandwidth requirements but stringent latency requirements: search term completions that arrive after the user has input additional characters are of no use.

To evaluate the usefulness of MapJAX in this context, we measured the average latency of completion retrieval (the *average request latency*) of both a MapJAX and standard AJAX implementation under a range of latency and bandwidth parameters. Specifically, we tested using bandwidth values ranging from 256Kb/s to 1024Kb/s, which are typical of home broadband connections, and latencies of both 20 and 70 ms, which correspond to close and average-distance servers. The standard AJAX version of the application made no attempt to prefetch, and we tested the MapJAX version with both prefetching enabled (using the English language policy described above) and prefetching disabled.

To measure average request latency, we used a workload generated by typing 65 search terms from the April/May 2006 Google Zeitgeist list of popular search terms into the AJAX version of the application, resulting in a trace of 423 completion requests. The completions returned were those that would have been returned by Google's version of the application (they were retrieved from Google and cached at our server ahead of time). The average number of suggestions per suggestion set was 5.71, and the average suggestion set size was 264 bytes. We discarded the first 10 observed latencies to avoid measuring noise due to the application loading.

The results of this test are shown in Figure 9. First,

we note that the non-MapJAX (i.e., standard AJAX) implementation average latencies and the MapJAX (no prefetching) average latencies are always within 3 ms of each other, indicating that MapJAX imposes minimal additional overhead in providing its programming model (even with the current unoptimized implementation). Second, we observe that when sufficient bandwidth is available, prefetching significantly decreases the average latency (by close to half in the 1024Kb/s case). As expected, when sufficient bandwidth is not available, prefetching delays the servicing of actual cache misses and hurts performance. Future work will include automatic network performance measurements to allow programmers to scale back or disable prefetching in these cases. Additional results (not shown) show negligible CPU or memory overhead for the MapJAX implementation relative to the non-MapJAX implementation.

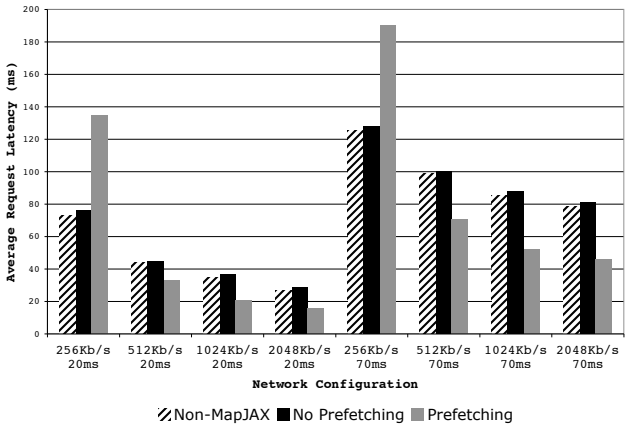

Figure 9: Average request latencies as generated by the auto-complete application on a sample workload under a range of simulated network conditions.

6.1.2 Mapping Application

In contrast to the auto-complete application, the mapping application represents a case in which the application is somewhat tolerant to latency (map tiles simply need to arrive before the user can scroll them completely off the screen) but has high bandwidth requirements.

To evaluate the utility of MapJAX in this context, we measured the average latency of map tile retrieval of both standard and MapJAX implementations under a range of bandwidth parameters (ranging from 256Kb/s to 8,192Kb/s) and a fixed latency of 70 ms. As before, the standard implementation made no attempt to prefetch, and we tested the MapJAX version with both prefetching enabled (using the OMNI policy described earlier, at various levels of prefetching) and prefetching disabled.

To measure the average request latency, we used a pair of user-generated workloads. Specifically, we asked a number of subjects to perform a simple navigation task using the mapping application with no bandwidth or delay constraints: scrolling from the MIT campus in Cambridge to the intersection of I-93 and MA-24 south of the city. The trace was formed by recording the GUI events (clicks and drags) thus generated. From this collection of traces, we chose two for testing. The first, which we call "hard," was generated by a user familiar with the area who was able to navigate at high speed. The second, which we call "easy," was generated by a user new to the area who navigated more slowly. The "hard" workload consisted of 467 GUI events resulting in 198 calls to access, and the "easy" workload consisted of 890 GUI events resulting in 196 calls to access. The average size of an image tile used by these workloads was 4,751 bytes.

Testing the application consisted of replaying these two traces and recording the access latency observed on non-canceled image tiles. Specifically, we used a viewport 8 tiles wide by 4 tiles high. The initial 32 images were loaded with prefetching disabled, and we did not record these latencies. We then enabled prefetching and replayed the trace. When computing average access latencies, we discarded the first 15 latencies generated by the trace to avoid measuring startup effects. The map tile images were those used by Google; they were downloaded and cached at our server ahead of time.

The results of these tests are presented in Figure 10 ("easy" trace) and Figure 11 ("hard" trace). Please note that the y-axis is log-scaled. To compensate for observed run-to-run variability, we report the average over three runs for each value, with error bars showing plus or minus one standard deviation.

We observe several interesting features of the graphs. First, at 256Kb/s, both workloads exhibit extremely high latencies with all implementations of the application, indicating that the 256Kb/s is insufficient bandwidth to support the workloads. Second, on the easy workload, MapJAX with prefetching disabled exhibits average latencies within 3% of the standard implementation except at 256Kb/s, where the average is highly variable and within 10%. On the hard workload, MapJAX with prefetching disabled exhibits average latencies within 12% of the standard implementation, except at 8192Kb/s, where it is within 17%.

These results indicate that the benefits of MapJAX are available with little overhead. Moreover, we believe that most of the overhead seen here is due to our unoptimized implementation and can be removed. Additional results (not shown) show that there is little to no startup overhead imposed by MapJAX, assuming that prefetching is disabled during startup. We also found MapJAX to impose negligible CPU or memory overheads relative to the

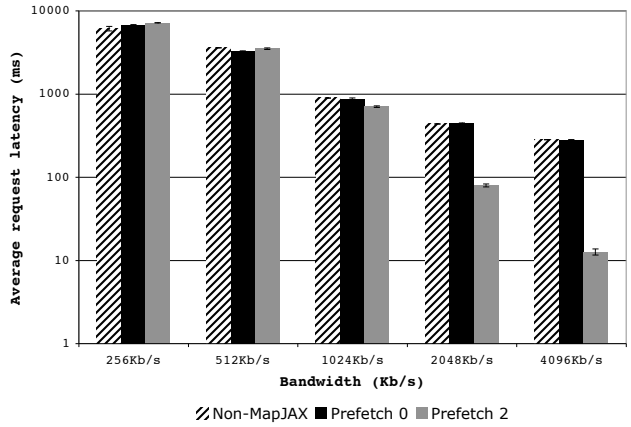

Figure 10: Results of running the mapping application with various prefetching policies on the "easy" workload using a simulated network with 70 ms latency and varied bandwidth constraints.

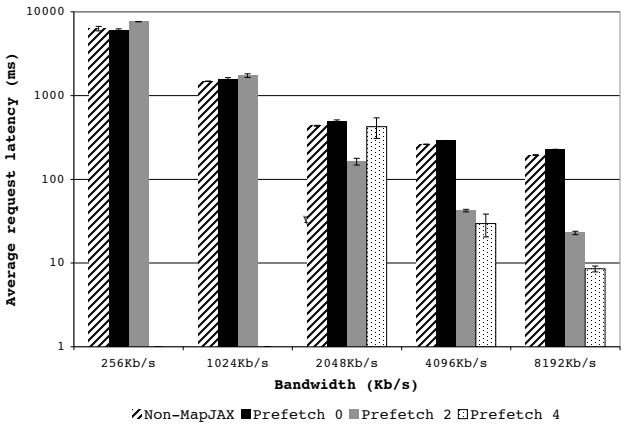

Figure 11: Results of running the mapping application with various prefetching policies on the "hard" workload using a simulated network with 70 ms latency and varied bandwidth constraints. The "prefetch 4" result is not shown for the 256Kb/s and 1024Kb/s cases as performance was extremely poor.

non-MapJAX implementation.

As in the auto-complete application, as spare bandwidth becomes available, prefetching is able to dramatically reduce the average access latency. The "hard" workload sees a 62.5% reduction with Omni-2 at 2048Kb/s and an 83.8% reduction in latency with Omni-2 at 4096Kb/s. The "easy" workload sees a 21.2% reduction with Omni-2 at 1024Kb/s, an 81.7% reduction with Omni-2 at 2048Kb/s, and a 95.5% reduction in latency with Omni-2 at 4096Kb/s. By contrast, prefetching increases access latency when spare bandwidth is not available, as would be expected; again, future work will allow programmers to scale back or disable prefetching in this case.

6.1.3 Mail

We exercise some features of the mail application in the microbenchmarks, below. Full-application tests provided no additional information beyond that obtained from the mapping and auto-complete applications and are omitted here.

6.2 Microbenchmarks

6.2.1 Parallel *For* loops and Request Combining

To evaluate parallel **for** loops and request combining, we measured the total time required to load a page of 40 message headers in the MapJAX implementation of our email application with both parallel and non-parallel **for** loops. In the parallel case, we tested with request combining both enabled and disabled. (Non-parallel **for** loops provide no opportunity for request combining, so we did not test that combination.) To eliminate network effects, we introduced no latency or bandwidth constraints and disabled prefetching.

The results of this experiment are shown in Table 2. Parallel **for** loops and request combining both provide a clear advantage: parallel **for** loops provide an order of magnitude speedup over non-parallel loops, as they are able to fetch all items required by the loop immediately, rather than waiting for one RPC to complete before sending the next. Request combining provides a speedup greater than a factor of two, as it cuts the number of RPCs issued from 40 to one. Had we used simulated network delays, the advantage of MapJAX would have been even greater.

For comparison, the non-MapJAX version of the code, which sends a single RPC for all 40 headers, averaged 77.6 ms over 10 trials with a standard deviation of 0.52. We believe most of the difference between that value and MapJAX parallel **for** loops with request combining is due to implementation artifacts and can be eliminated.

	Combining	**No Combining**
pfor	117.8 ± 5.51	277.2 ± 12.18
for	*(Not Run)*	1465.0 ± 18.90

Table 2: Effectiveness of *pfor* loops and request combining. Values shown are the times to load a page of 40 message headers in our email application, averaged over 10 trials and given in milliseconds $\pm$ one standard deviation.

6.2.2 Locks

To evaluate the overhead of our lock implementation, we compared the performance of the MapJAX version of the mapping application on the "hard" trace against the performance of a version that manually ordered updates instead of using locks. To eliminate network effects, we added no bandwidth or delay constraints and disabled prefetching. Averaging over three runs, the version using locks had an average access latency of 59.76 ms, while the version using manual ordering had an average access latency of 58.20 ms. Standard deviations were 1.82 and 1.80, respectively. We conclude that our lock implementation adds negligible overhead.

6.2.3 Request Canceling

To evaluate the contribution made by request canceling, we modified the MapJAX version of our mapping application to not cancel requests and ran the "easy" workload on a simulated 256Kb/s, 70ms network with prefetching disabled. (This simulated network provides insufficient bandwidth to support the trace, and thus request canceling is particularly important). The average request latency (averaged over three runs) was 33,397.92 ms, which is far greater than the 6605.40 ms obtained with canceling enabled. Additional experiments (not shown) found the importance of canceling to decrease as bandwidth increased, which is the expected result: in an environment where bandwidth is plentiful, there is no need to conserve it. We conclude that request canceling brings performance benefits worthy of the additional complexity it adds to the model.

7 Related Work

The trade-off between threading and event-based models has been well studied, recently in [14], which considered the issue in a server context. The MapJAX compiler is similar in some respects to TAME [7], which carries out a similar continuation-elimination function for code written using the libasync [8] C++ library for event-driven network servers.

To our surprise, we were unable to find previous work on locks with RLU semantics. In [6], the authors propose a lock reservation scheme to decrease the overhead of lock acquisition in Java VMs, their scheme only allows one thread to hold a lock reservation, whereas in our scheme multiple threads can reserve a lock. Our locks might appear similar to callback-based, asynchronously-acquired locks (e.g. as in [10]) but in fact they provide stronger semantics. Specifically, we guarantee that statements between the calls to reserve and lock will execute strictly before any statements after the call to lock. By contrast, an asynchronously-acquired lock makes no such guarantee about when its callback will be executed.

There have been several proposals of programming models for writing rich web applications, from ambitious efforts that provide fresh display layout and programming languages to smaller, lighter-weight efforts that try to smooth the rough edges of the current model. To our knowledge no system exists that provides the shared data abstraction, elimination of callbacks, and array of performance optimizations available in MapJAX.

Examples of ambitious web programming systems include Java applets and Adobe Flash. Such systems have the advantage of starting from a clean slate, which allows them to ignore the imperfections of the standard web development model. However, on the Internet, incremental deployability is often key: technologies that require users to install new software and developers to learn new languages often do not succeed. Additionally, systems of this type can be difficult to integrate cleanly into HTML-based pages, which renders them unattractive from a designer's perspective. Flash comes closest to the MapJAX programming model of any of the current systems: it provides data structure objects that can be bound to server-side data. Flash lacks MapJAX's support for prefetching, parallel **for**, and RLU locks, and it does not eliminate callbacks. Additionally, it requires a separate browser plugin to run and requires the programmer to learn an additional language.

At the other end of the spectrum, the growth in popularity of AJAX has given rise to numerous small libraries that attempt to put a friendlier face on AJAX development, including Prototype [13], Mochikit [10], JSON-RPC-Java [9], and Direct Web Remoting [15]. In general, these libraries tend to provide some subset of three classes of features.

First, some offer a set of reusable user interface controls for AJAX applications, such as a table that is dynamically filled in with data from the server. While such controls are similar in spirit to MapJAX shared data structures, they are hardly a full-fledged programming model. Second, some libraries attempt to resolve some of the deficiencies in the Javascript programming language; e.g., they might add extra functions for handling strings, accessing HTML elements, managing asynchronous tasks, or logging. Finally, some libraries include support for some variant of RPC built on top of AJAX requests. The level of RPC abstraction provided is widely variable: the Prototype and Mochikit frameworks free the programmer from some of the event-handling associated with managing AJAX requests but keep the asynchronous HTTP-request model, whereas DWR and JSON-RPC-Java both extend Java RMI [16] to the browser. In all cases, however, these libraries at best provide a more pleasant interface over what is essentially

an asynchronous RPC call, along with problems of that abstraction.

In between the two above extremes are web development platforms such as the Google Web Toolkit [3], Ruby on Rails (RoR) [4] and OpenLaszlo [11]. These systems use existing browser technologies to deploy their applications, but they provide a higher level of abstraction than the small libraries discussed above. The Google Web Toolkit provides a Java-to-AJAX compiler, but it does not include MapJAX-style shared data structures, and it provides callback-based RPCs. Ruby on Rails is a rapid development framework for database-backed applications and thus includes some aspects of a data model, but it supports neither prefetching nor callback elimination. Finally, OpenLaszlo provides a compiler from their own language to AJAX, although again without shared data structures, and it retains asynchronous callbacks.

8 Conclusion

This paper has presented MapJAX, a new programming environment for AJAX-style web applications. In contrast to current systems based on asynchronous RPCs, MapJAX provides application programmers with the abstraction of logical data structures shared between client and server, accessed through a familiar programming model based on objects, threads, and locks. MapJAX also includes a number of additional features (parallel **for** loops, data prefetching, and request canceling) that help programmers implement highly-responsive applications.

There are several areas of MapJAX that warrant future work. First, we would like the system to better adapt to changing network conditions: the runtime should characterize the performance of the network and adapt prefetching accordingly, either automatically or by exposing this information to the application. It would also be helpful to have a more intelligent way to group fetches into RPCs: as mentioned in Section 4.2.5, there are tradeoffs involved in choosing message size. Additionally, the cache could be extended to persist on disk across reloads of the application. Finally, we would like to extend MapJAX to handle writable data structures.

In summary, we have implemented three prototypical AJAX applications using both standard AJAX techniques and MapJAX. We tested them under realistic Internet latency and bandwidth constraints and found that the unoptimized MapJAX versions of the applications performed comparably to the standard AJAX versions, and that MapJAX performance optimizations could dramatically improve performance, by up to a factor of 2 in some cases. Finally, we have performed microbenchmarks exercising each of the performance optimizations we provide and have shown the contributions made by each. We believe our results show that MapJAX meets its goals of reducing the development complexity while simultaneously improving the performance of AJAX web applications.

9 Acknowledgments

The authors thank Robert Morris and Emil Sit for valuable conversations during the development of this system, as well as the anonymous reviewers for their comments.

References

[1] FESTA, P. Will AJAX help google clean up? http://news.com.com/Will+AJAX+help+Google+clean+up/2100-1032_3-5621010.html, March 2005.

[2] GARRETT, J. Ajax: a new approach to web applications, February 2005. http://www.adaptivepath.com/publications/essays/archives/000385.php.

[3] GOOGLE, INC. Google web toolkit. http://code.google.com/webtoolkit.

[4] HASSON, D. Ruby on Rails. http://rubyonrails.org.

[5] JSON. http://www.json.org.

[6] KAWACHIYA, K., KOSEKI, A., AND ONODERA, T. Lock reservation: Java locks can mostly do without atomic operations. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2002), ACM Press, pp. 130–141.

[7] KROHN, M., KOHLER, E., AND KAASHOEK, M. F. Simplified event programming for busy network applications. In *Proceedings of the 2007 USENIX Annual Technical Conference* (Santa Clara, CA, USA, June 2007), USENIX.

[8] MAZIERES, D. A toolkit for user-level file systems. In *Proceedings of the 2001 USENIX Annual Technical Conference* (Boston, MA, June 2001), USENIX.

[9] METAPARADIGM PTE LTD. JSON-RPC-Java. http://oss.metaparadigm.com/jsonrpc/.

[10] Mochi Media, LLC. http://mochikit.com.

[11] Open Laszlo. http://openlaszlo.org.

[12] RIZZO, L. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review 27*, 1 (1997), 31–41.

[13] STEPHENSON, S. Prototype JavaScript Library. http://prototype.conio.net.

[14] VON BEHREN, R., CONDIT, J., AND BREWER, E. Why events are a bad idea for high-concurrency servers, 2003.

[15] WALKER, J., AND GOODWIN, M. DWR - Easy AJAX for Java. http://getahead.ltd.uk/dwr.

[16] WOLLRATH, A., RIGGS, R., AND WALDO, J. A distributed object model for the Java system. In *2nd Conference on Object-Oriented Technologies & Systems (COOTS)* (1996), USENIX Association, pp. 219–232.

[17] ZEUS TECHNOLOGY LIMITED. Zeus Web Server. http://www.zeus.co.uk.