Crom: Faster Web Browsing Using Speculative Execution

James Mickens, Jeremy Elson, Jon Howell, and Jay Lorch
Microsoft Research
mickens,jelson,jonh,lorch@microsoft.com

Abstract

Early web content was expressed statically, making it amenable to straightforward prefetching to reduce user-perceived network delay. In contrast, today's rich web applications often hide content behind JavaScript event handlers, confounding static prefetching techniques. Sophisticated applications use custom code to prefetch data and do other anticipatory processing, but these custom solutions are costly to develop and application-specific.

This paper introduces Crom, a generic JavaScript speculation engine that greatly simplifies the task of writing low-latency, rich web applications. Crom takes preexisting, non-speculative event handlers and creates speculative versions, running them in a cloned browser context. If the user generates a speculated-upon event, Crom commits the precomputed result to the real browser context. Since Crom is written in JavaScript, it runs on unmodified client browsers. Using experiments with speculative versions of real applications, we show that pre-commit speculation overhead easily fits within user think time. We also show that speculatively fetching page data and precomputing its layout can make subsequent page loads an order of magnitude faster.

1 Introduction

With the advent of web browsing, humans began a new era of waiting for slow networks. To reduce user-perceived download latencies, researchers devised ways for browsers to prefetch content and hide the fetch delay within users' "think time" [4, 15, 17, 20, 23]. Finding prefetchable objects was straightforward because the early web was essentially a graph of static objects stitched together by declarative links. To discover prefetchable data, one merely had to traverse these links.

In the web's second decade, static content graphs have been steadily replaced by *rich Internet applications* (RIAs) that mimic the interactivity of desktop applications. RIAs manipulate complex, time-dependent server-side resources, so their content graphs are dynamic. RIAs also use client-side code to enhance interactivity. This eliminates the declarative representation of the content graph's edges, since now content can be dynamically named and fetched in response to the execution of an imperative event handler.

1.1 New Challenges to Latency Reduction

RIAs introduce three impediments to reducing user-perceived browser latencies. First, prefetching opportunities that once were statically enumerable are now hidden behind imperative code such as event handlers. Since event handlers have side effects that modify application state, they cannot simply be executed "early" to trigger object fetches and warm the browser cache.

Second, user inputs play a key role in naming the content to fetch. These inputs may be as simple as the clicking of a button, or as unconstrained as the entry of arbitrary text into a search form. Given the potentially combinatorial number of objects that are nameable by future user inputs, a prefetcher must identify a promising *subset* of these objects that are likely to be requested soon.

Third, RIAs spend a non-trivial amount of time updating the screen. Once the browser has fetched the necessary objects, it must devise a layout tree for those objects and render the tree on the display. For modern, graphically intensive applications, screen updates can take hundreds of milliseconds and consume 40% of the processor cycles used by the browser [22]. Screen updates contribute less to page load latencies than network delays do, but they are definitely noticeable to users. Unfortunately, warming the browser cache before a page is loaded will not reduce its layout or rendering cost.

1.2 Prior Solutions

To address these challenges, some RIAs use custom code to speculate on user intent. For example, email clients may prefetch the bodies of recently arrived messages, or speculatively upload attachments for emails that have not yet been sent. Photo gallery applications often prefetch large photos. Online maps speculatively download new map tiles that may be needed soon. The results page for a web search may prefetch the highest ranked targets to reduce their user-perceived load time. While such speculative code provides the desired latency reductions, it is often difficult to write and tightly integrated into an application's code, making it impossible to share across applications.

1.3 Our Solution: Crom

To ease the creation of low-latency web applications, we built Crom, a reusable framework for speculative

JavaScript execution. In the simplest case, the Crom API allows applications to mark individual JavaScript event handlers as *speculable*. For each such handler `h`, Crom creates a new version `h_shadow` which is semantically equivalent to `h` but which updates a shadow copy of the browser state. During user think-time, e.g., when the user is looking at a newly loaded page, Crom makes a shadow copy of the browser state and runs `h_shadow` in that context. As `h_shadow` runs, it fetches data, updates the shadow browser display, and modifies the application's shadow JavaScript state. Once `h_shadow` finishes, Crom stores the updated shadow context; this context includes the modified JavaScript state as well as the new screen layout. Later, if the user actually generates the speculated-upon event, Crom commits the shadow context. In most cases, the commit operation only requires a few pointer swaps and a screen redraw. This is much faster than synchronously fetching the web data, calculating a new layout, and rendering it on the screen.

To constrain the speculation space for arbitrarily-valued input elements, applications use *mutator functions*. Given the current state of the application, a mutator generates probable outcomes for an input element. For example, an application may provide an autocompleting text box which displays suggested words as the user types. Once a user has typed a few letters, e.g., "red", the application passes Crom a mutator function which modifies fresh shadow domains to represent appropriate speculations; in this example, the mutator may set one shadow domain's text box to "red sox", and another domain's text box to "red cross". Later, when the user generates an actual event for the input, Crom uses application-defined *equivalence classes* to determine whether any speculative domain is the outcome for the actual event and thus appropriate to commit.

The Crom API contains additional functionality to support speculative execution. For example, it provides an explicit AJAX cache to store prefetched data that the regular browser cache would ignore. It also provides a server-side component that makes it easier to speculatively upload client data. Taken as a whole, the Crom library makes it easier for developers to reason about asynchronous speculative computations.

1.4 Our Contributions

This paper makes the following contributions:

- We identify four sources of delay in rich web applications, explaining how they can be ameliorated through speculative execution (§4).
- We describe an implementation of the Crom API which is written in standard JavaScript and runs on unmodified browsers (§5). The library, which is 65 KB in size, dynamically creates new browser contexts, rewrites event handlers to speculatively execute inside of these contexts, and commits them when appropriate.
- We describe three implementation optimizations that mitigate JavaScript-specific performance limitations on speculative execution (§5).
- Using these optimizations, we demonstrate the feasibility of browser speculation by measuring the performance of three modified applications that use the Crom library. We show that Crom's pre-commit speculation overhead is no worse than 114 ms, making it feasible to hide speculative computation within user think time. We also quantify Crom's reduction of user-perceived latency, showing that Crom can reduce load times by an order of magnitude under realistic network conditions (§6).

By automating the low-level tasks that support speculative execution, Crom greatly reduces the implementation effort needed to write low-latency web applications.

2 Background: Client-side Scripting

The language most widely used for client-side scripting is JavaScript [7], a dynamically typed, object-oriented language. JavaScript interacts with the browser through the *Document Object Model* (DOM) [24], a standard, browser-neutral interface for examining and manipulating the content of a web page. Each element in a page's HTML has a corresponding object in the *DOM tree*. This tree is a property of the `document` object in the global `window` name space. The browser exposes the DOM tree as a JavaScript data structure, allowing client-side applications to manipulate the web page by examining and modifying the properties of DOM JavaScript objects, often referred to as *DOM nodes*. The DOM allows JavaScript code to find specific DOM nodes, create new DOM nodes, and change the parent-child relationships among DOM nodes.

A JavaScript programmer can create other objects besides DOM nodes. These are called *application heap* objects. All non-primitive objects, including functions, are essentially dictionaries that maps property names to property values. A property value is either another object or a primitive such as a number or boolean. Properties may be dynamically added to, and deleted from, an object. The `for-in` construct allows code to iterate over an object's property names at run-time.

Built-in JavaScript objects like a `String` or a DOM node have *native code implementations*— their methods are executed by the browser in a way that cannot be introspected by application-level JavaScript. In contrast, JavaScript *can* fetch the source code of a user-defined method by calling its `toString()` method.

JavaScript uses *event handlers* to make web pages interactive. An event handler is a function assigned to a special property of a DOM node; the browser will invoke that property when the associated event occurs.[1] For example, when a user clicks a button, the browser will invoke the `onclick` property of that button's DOM object. This gives application code an opportunity to update the page in response to user activity.

The AJAX interface [9] allows a JavaScript application to asynchronously fetch web content. AJAX is useful because JavaScript programs are single-threaded and network operations can be slow. To issue an AJAX request, JavaScript code creates an `XmlHTTPRequest` object and assigns an event handler to its `onreadystatechange` property. The browser will call this handler when reply data arrives.

3 Design

Crom's goal is to reduce the developer effort needed to create speculative applications. In particular, Crom tries to minimize the amount of *custom code* that developers must write to speculate on *user inputs* like keyboard and mouse activity. Crom's API leverages the fact that event-driven applications already have a natural grammar for expressing speculable user actions—each action can be represented as an event handler and a particular set of arguments to pass to the handler. Using the Crom API, applications can mark certain actions as speculable. Crom will then automatically perform the low-level tasks needed to run the speculative code paths, isolate their side-effects, and commit their results if appropriate.

Crom provides an *application-agnostic* framework for speculative execution. This generality allows a wide variety of programs to benefit from Crom's services. However, as a consequence of this generality, Crom requires some developer guidance to ensure correctness and to prevent speculative activity from consuming too many resources. In particular:

- Speculating on all possible user inputs is computationally infeasible. Thus, Crom relies on the developer to constrain the speculation space and suggest reasonable speculative inputs for a particular application state (§4.1.1 and §5.1.3).
- Speculative execution should only occur when the application has idle resources, e.g., during user think time. Crom's speculations are explicitly initiated by the developer, and Crom trusts the developer to only issue speculations when resources would otherwise lie fallow.
- A speculative context should only be committed if it represents a realizable outcome for the current application state. In particular, the *initial* state of a committing speculative context must have been equal to the *current* state of the application. This guarantees that the speculative event handler did the same things that its non-speculative version would do if executed now. We call this safety principle *start-state equivalence*. Crom could automatically check for this in an application-agnostic way by bit-comparing the current browser state with the initial state for the speculative context. However, performing such a comparison would often be expensive. Thus, Crom requires the developer to leverage application knowledge and define an equivalence function that determines whether a speculative context is appropriate to commit for a given application state (§4.1.1 and §5.1.3).
- A client-side event handler may generate writes to client-side state *and* server-side state. The developer must ensure that client-side speculation is read-only with respect to server state, or that server-side updates issued by speculative code can be undone, e.g., by resetting the "message read" flags on an email server (§ 4.1.2).

Writing speculative code is inherently challenging. Crom hides some of the implementation complexity, but it does not completely free the developer from reasoning about speculative operations. We believe that Crom strikes the appropriate balance between the competing tensions of correctness, ease of use, and performance.

Crom ensures correctness by rewriting speculative code to only touch shadow state, and by using developer-defined equivalence functions to determine commit safety. Crom provides ease of use through its generic speculation API. With respect to performance, Crom's goal is *not* to be as CPU-efficient as custom speculative code. Indeed, Crom's speculative call trees will generally be slower than hand-crafted speculation code. Such custom code has no rewriting or cloning overhead, and it can speculate in a targeted way, eliding the code in an event handler call chain that is irrelevant to, say, warming a cache. However, Crom's speculations only need to be "fast enough"—they must fit within user think time, be quick with respect to network latencies, and not disturb foreground computations. If these conditions are satisfied, Crom's computational inefficiency relative to custom speculation code will be moot, and Crom's speculations will mask essentially as much network latency as hand-coded speculations would.

In addition to processor cycles, speculative activity requires network bandwidth to exchange data with servers. Crom does not seek to reduce this inherent cost of speculation. As with hand-crafted speculative solutions, developers must be mindful of network overheads and be judicious in how many Crom speculations are issued.

Figure 1 lists the primary Crom API. We discuss this API in greater detail in the next two sections. Most of the technical challenges lie with the implementation of `Crom.makeSpeculative()`, which allows an application to define speculable user actions.

[1]This description applies to the DOM Level 0 event model. The DOM Level 2 model is also common, but it has incompatible semantics across browsers. We do not discuss it in this paper.

`Crom.makeSpeculative(DOMnode,eventName, mutator,mutatorArgs,stateSketch, DOMsubtree)`	Register `DOMnode.eventName` as a speculable event handler (§4.1 and §5.1). The `mutator`, `mutatorArgs`, and `stateSketch` arguments constrain the speculation space and define the equivalence classes for commits (§4.1.1 and §5.1.3). `DOMsubtree` defines a speculation zone (§5.5.2).
`Crom.autoSpeculate`	Boolean which determines whether Crom should automatically respeculate after committing a prior speculation (§4.1.1).
`Crom.forceSpeculations()`	If `autoSpeculate` is `false`, this method forces Crom to issue pending speculations.
`Crom.maxSpeculations(N)`	Limits number of speculations Crom issues (§5.6).
`Crom.createContextPool(N, stateSketch, DOMsubtree)`	Proactively make `N` speculative copies of the current browser context; tag them using the `stateSketch` function (§5.5.3).
`Crom.rewriteCG(f)`	Rewrite a closure-generating function to make it amenable to speculation (§5.1.5).

`Crom.prePOST(formInput, specUploadDoneCallback)`	Collaborates with Crom's server-side component to make a form element speculatively upload data (§4.2 and §5.4).

`Crom.cacheAdd(key, AJAXdata)` `Crom.cacheGet(key)`	Used to cache speculatively fetched AJAX data that would otherwise be ignored by the regular browser cache (§4.1.2 and §5.3).

Figure 1: Crom API calls and configuration settings.

4 Speculation Opportunities & Techniques

In this section, we describe four ways that speculative execution can reduce latency in rich Internet applications. We also explain how to leverage the Crom API from Figure 1 to exploit these speculative opportunities.

4.1 Simple Prefetching

When a user triggers a heavyweight state change in a RIA, the browser does four things. First, it executes JavaScript code associated with an event handler. In turn, this code fetches data from the local browser cache or external web servers. Once the content is fetched, the browser determines the new display layout for the page. Finally, the browser draws the content on the screen.

Pulling data across the network is typically the slowest task, so warming the browser cache by speculatively prefetching data can dramatically improve user-perceived latencies. For example, a photo gallery or an interactive map application can prefetch images to avoid synchronous fetches through a high-latency or low-bandwidth network connection. As another example, consider the DHTMLGoodies tab manager [12], which allows applications to create tab pane GUIs. When the user clicks a "new tab" link, the tab manager issues an AJAX request to fetch the new tab's content. When the AJAX request completes, a callback dynamically creates a new display tab and inserts the returned HTML into the DOM tree. Figure 2 demonstrates how to make this operation speculative. When the application calls `Crom.makeSpeculative()`, Crom automatically makes a shadow copy of the current browser state and rewrites the `onclick` event handler, creating a speculative version that accesses shadow state. When the application calls `Crom.forceSpeculations()`, Crom runs the rewritten handler in the hidden context, fetching the AJAX data and warming the real browser cache.

```
<div id='tab-container'>
   <div class='dhtmlgoodies_aTab'>
      This is the initial tab.
      <a href='#' id='loadLink'>
          Click to load new tab.
      </a>
   </div>
</div>
<script>
var link = document.getElementById('loadLink');
link.onclick = function(){
   //Invoke DHTMLGoodies API to make new tab
   createNewTab('http://www.foo.com');
};
Crom.makeSpeculative(link, 'onclick');
Crom.forceSpeculations();
</script>
```

Figure 2: Creating a new tab GUI element.

Once the browser cache has been warmed, Crom can discard the speculative context. However, saving the context for future committing provides greater reductions in user-perceived fetch latencies (§4.3).

4.1.1 Speculating on Multi-valued Inputs

In the previous example, an application speculated on an input element with one possible outcome, i.e., being clicked. Other input types can generate multiple speculable outcomes. For example, a list selector allows one of several options to be chosen, and a text box can accept arbitrary character inputs.

To speculate on a multi-valued input, an application passes a *mutator function*, a *state sketch function*, and a vector of N *mutator argument sets* to `Crom.makeSpeculative()`. Crom makes N copies of the current browser state, and for each argument set, Crom runs the rewritten mutator with that argument set in the context of a shadow browser environment. This generates N distinct contexts, each representing a different speculative outcome for the input element. Crom passes each context to the sketch function, generating an application-defined signature string for that context. Crom tags each context with its unique sketch and then runs the speculative event handler in each context, saving the modified domains. Later, when the user actually generates an event, Crom determines the sketch for the current, non-speculative application state. If Crom has a speculative context with a matching tag, Crom can safely commit that context due to start-state equivalence.

Figure 3 shows an example of how Crom speculates on multi-valued inputs. In this example, we use the autocompleting text box from the popular script.aculo.us JavaScript library [21]. The first two lines of HTML define the text input and the button which triggers a data fetch based on the text string. We create a custom JavaScript object called `acManager` to control the autocompletion process and register it with the script.aculo.us library. The library invokes `acManager.customSelector()` whenever the user generates a new input character. Once the user has typed five characters, `acManager` uses AJAX to speculatively fetch the data associated with each suggested autocompletion. The state sketch function simply returns the value of the search text—a speculative context is committable if its search text is equivalent to that of the real domain.

Note that the code sets `Crom.autoSpeculate` to `false`, indicating that Crom should not automatically respeculate on the handler when a prior speculation for the handler commits. The autocompletion logic explicitly forces speculation when the user has typed enough text to generate completion hints.

```
<input id='sText' type='text' />
<button id='sButton'>Search</button>
<script>
var sText = document.getElementById('sText');
var sButton = document.getElementById('sButton');
sButton.onclick = function(){
   updateDisplayWithAJAXdata(sText.value);
};

Crom.autoSpeculate = false;
function mutator(arg){ //Set value of text input
   sText.value = arg;
}
function sketch(ctx){
   var doc = ctx.document;
   var textInput = doc.getElementById('sText');
   return textInput.value;
}
var acManager = {
   getHints: function(textSoFar){
      //Logic for generating autcompletions;
      //returns an array of strings
   },
   customSelector: function(textSoFar){
      if(textSoFar.length == 5){
         var hints = this.getHints(textSoFar);
         Crom.makeSpeculative(sButton,
            'onclick',mutator,hints,sketch);
         Crom.forceSpeculations();
      }
      //Display autocompletions to user...
   }
};
new Autocompleter('sText', acManager);
</script>
```

Figure 3: Speculating on an autocompletion event.

4.1.2 Separating Reads and Updates

In some web applications, pulling data from a server has side effects on the client *and* the server. In these situations, speculative computations must not disturb foreground state on either host. For example, the Decimail webmail client [5] uses AJAX to wrap calls to an IMAP server. The `fetchNewMessage()` operation updates client-side metadata (e.g., a list of which messages have been fetched) and server-side metadata (e.g., which messages should be marked as seen).

To speculate on such a read/write operation, the developer must explicitly decompose it into a read portion and a write portion with respect to server state. For example, to add Crom speculations to the Decimail client, we had to split the preexisting `fetchNewMessage()` operation into a read-only `downloadMessage()` and a metadata-writing `markMessageRead()`. The read-only operation downloads an email from the server, but specifies in the IMAP request that the server should not mark the message as seen. The `markMessageRead()` tells the server to update this flag, effectively committing the message fetch on the server-side. Inside `fetchNewMessage()`, the call

```
<form action='server.com/recv.py' method='post'>
   <div>
      <label>File 1:</label>
      <input type='file' id='fInput'/>
   </div>
   <div>
      <input type='submit' value='Send data!'/>
   </div>
</form>

<script>
var fInput = document.getElementById('fInput');
Crom.speculativePOST(fInput,
           function(){alert('File uploaded!')};
</script>
```

Figure 4: Making a POST operation speculative.

to `markMessageRead()` is conditioned on whether `fetchNewMessage()` is running speculatively; code can check whether this is true by reading the special `Crom.isSpecEx` boolean.

Although `downloadMessage()` may be read-only with respect to the server, it may update client-side JavaScript state. So, when speculating on `fetchNewMessage()`, we run `downloadMessage()` in a speculative execution context. In speculative or non-speculative mode, `downloadMessage()` places AJAX responses into a new cache provided by Crom. Later, when `downloadMessage()` runs in non-speculative mode, it checks this cache for the message and avoids a refetch from the server.

Like the regular browser cache, Crom's AJAX cache persists across speculations (although not application reloads). The regular cache will store AJAX results containing "expires" or "cache control" headers [6], so an application-level AJAX cache may seem superfluous. However, some AJAX servers do not provide caching headers, making it impossible to rely on the regular cache to store AJAX data if the client side of the application is developed separately from the server side. Examples of such scenarios include mash-ups and aggregation sites.

4.2 Pre-POSTing Uploads

Prefetching allows Crom to hide download latency. However, in some situations, such as Decimail's attach-file function, the user is stalled by upload (HTTP POST) delays. To hide this latency, Crom's client and server components cooperate to create a *POST cache*. When a user specifies a file to send, Crom speculates that the user will later commit the send. Crom asynchronously transfers the data to the server's POST cache. Later, if the user commits the send, the asynchronous POST will be finished (or at least already in-progress).

Figure 4 demonstrates how to make a POST operation speculative. The web application simply registers the relevant `input` element with the Crom library. Once the user has selected a file, Crom automatically starts uploading it to the server. When the speculative upload completes, Crom invokes an optionally provided callback function; this allows the application to update foreground (i.e., non-speculative) GUI state to indicate that the file has safely reached the server.

Speculative uploading is not a new technique, and it is used by several popular services like GMail. Crom's contribution is providing a generic framework for adding speculative uploads to non-speculative applications.

4.3 Saving Client Computation

When an application updates the screen, the browser uses CPU cycles for *layout* and *rendering*. During layout, the browser traverses the updated DOM tree and determines the spatial arrangement of the elements. During rendering, the browser draws the laid-out content on the screen. Speculative cache warming can hide fetch latency, but it cannot hide layout or rendering delays.

Crom stores each speculative browser context inside an invisible `<iframe>` tag. As a speculative event handler executes, it updates the layout of its corresponding `iframe`. When the handler terminates, Crom saves the already laid-out `iframe`. Later, if the user generates the speculated-upon event, Crom commits the speculative DOM tree in the `iframe` to the live display, paying the rendering cost but avoiding the layout cost. The result is a visibly smoother page load.

4.4 Server Load Smoothing

Some client delays are due not to network delays, but to congestion at the server due to spiky client loads. Using selective admission control at the server, speculative execution spreads client workload across time, just as speculation plus differentiated network service smooths peak network loads [3]. When the server is idle, speculations slide requests earlier in time, and when the server is busy, speculative requests are rejected and the associated load remains later in time. This paper does not explore server smoothing further, but the techniques described above, together with prioritized admission control at the server, should adequately expose this opportunity.

5 Implementation

The client-side Crom API could be implemented inside the browser or by a regular JavaScript library. For deployability, we chose the latter option. In this section, we describe our library implementation and the optimizations needed to make it performant.

5.1 Making Event Handlers Speculative

To create a speculative version of an event handler bound to DOM node `d`, an application calls `Crom.makeSpeculative(d, eventName)`; Figures 2 and 3 provide sample invocations of this function.

The `makeSpeculative()` method does two things. First, it creates a shadow browser context for the speculative computation. Second, it creates a new event handler that performs the same computation as the original one, but reads and writes from the speculative context instead of the real one. We discuss context cloning and function rewriting in detail below.

5.1.1 The Basics of Object Copying

Crom clones different types of objects using different techniques. For primitive values, Crom just returns the value. For built-in JavaScript objects like `Dates`, Crom calls the relevant built-in constructor to create a semantically equivalent but referentially distinct object.

JavaScript functions are first-class objects. Calling a function's `toString()` method returns the function's source code. To clone a function `f`, Crom calls `eval(f.toString())`, using the built-in `eval()` routine to parse the source and generate a semantically equivalent function. Like any object, `f` may have properties. So, after cloning the executable portion of `f`, Crom uses a `for-in` loop to discover `f`'s properties, copying primitives by value and objects using deep copies.

To clone a non-function object, Crom creates an initially empty object, finds the source object's properties using a `for-in` loop, and copies them into the target object as above. Since object graphs may contain cycles, Crom uses standard techniques from garbage collection research [11] to ensure that each object is only copied once, and that the cloned object graph is isomorphic to the real one.

To clone a DOM tree with a root node `n`, Crom calls the native DOM method `n.cloneNode(true)`, where the boolean parameter indicates that `n`'s DOM children should be recursively copied. The `cloneNode()` method does not copy event handlers or other application-defined properties belonging to a DOM node. Thus, Crom must copy these properties explicitly, traversing the speculative DOM tree in parallel with the real one and updating the properties for each speculative node. Non-event-handler properties are deep-copied using the techniques described above. Since Crom rewrites handlers and associates special metadata with them, Crom assumes that user-defined code does not modify or introspect event handlers. So, Crom shallow-copies event handlers by reference.

5.1.2 Cloning the Entire Browser State

To clone the whole browser context, Crom first copies the real DOM tree. Crom then creates an invisible `<iframe>` tag, installing the cloned DOM tree as the root tree of the `iframe`'s `document` object. Next, Crom copies the application heap, which is defined as all JavaScript objects in the global namespace and all objects reachable from those roots. Crom discovers the global properties using a `for-in` loop over `window`. Crom deep-copies each of these properties and inserts the cloned versions into an initially empty object called `specContext`. `specContext` will later serve as the global namespace for a speculative execution.

Global properties can be referenced with or without the `window.` prefix. To prevent `window.globalVar` from falling through to the real `window` object, Crom adds a property to `specContext` called `window` that points to `specContext`. Crom also adds a `specContext.document` property that points to the hidden `<iframe>`'s `document` object. As we explain in Section 5.1.4, this forces DOM operations in the speculative execution to touch the speculative DOM tree instead of the real one.

5.1.3 Commit Safety & Equivalence Classes

As described so far, a shadow context is initialized to be an exact copy of the browser state at clone time. This type of initialization has an important consequence: it prevents us from speculating on user intents that are not an immediate extension of the current browser state. For example, a text input generates an `onchange` event when the user types some characters and then shifts input focus to another element. If the text input is empty when Crom creates a speculative domain, Crom can speculatively determine what the `onchange` handler would do when confronted with an empty text box. However, if Crom creates shadow contexts as exact copies of the current browser state, Crom has no way to speculate on what the handler would do if the user had typed, say, "valhalla" into the text input.

To address this problem, we allow applications to provide three additional arguments to `Crom.makeSpeculative()`: a *mutator function*, a *mutator argument vector*, and a *state sketch function*. Section 4.1.1 provides an overview of these parameters. Here, we only elaborate on the sketch function and its relationship to committability.

The sketch function accepts a global namespace, speculative or real, and returns a unique string identifying the salient application features of that name space. Each speculative context is initially a perfect copy of the real browser context at time t_0. Thus, at t_0, before any speculative code has run, the new speculative context has the same sketch as the real context. Crom tags each speculative context with the state sketch of its source context. Later, at time t_1, when Crom must decide whether the speculative context is committable, it calculates the state sketch for the real browser context at t_1. A speculative context is only committable if its sketch tag matches the sketch for the current browser context. This ensures that the speculative context started as a semantically equivalent copy of the current browser state, and therefore represents the appropriate result for the user's new input.

State sketches provide a convenient way for applications to map semantically identical but bit-different browser states to a single speculable outcome. For example, the equivalence function in Figure 3 could canonicalize the search strings `blue\tbook` and `blue\t\tbook` to the same string.

5.1.4 Rewriting Handlers

After creating a speculative browser context, Crom must create a speculative version of the event handler, i.e., one that is semantically equivalent to the original but which interacts with the speculative context instead of the real one. To make such a function, Crom employs JavaScript's `with` statement. Inside a `with(obj){...}` statement, the properties of `obj` are pushed to the front of the name resolution chain. For example, if `obj` has a property `p`, then references to `p` touch `obj.p` rather than a globally defined `p`.

To create a speculative version of an event handler, Crom fetches the handler's source code by calling its `toString()` method. Next, Crom alters the source code string, placing it inside a `with(specContext)` statement. Finally, Crom uses `eval()` to generate a compiled function object. When Crom executes the new handler, each handler reference to a global property will be directed to the cloned property in `specContext`.

The `with()` statement binds lexically, so if the original event handler calls other functions, Crom must rewrite those as well. Crom does this lazily: for every function or method call `f()` inside the original handler, Crom inserts a new variable declaration `var __rewritten_f = Crom.rewriteFunction(f, specContext);`, and replaces calls to `f()` with calls to `__rewritten_f()`.

The `document` object mediates application access to the DOM tree. `specContext.document` points to the shadow DOM tree, so speculative DOM operations can only affect speculative DOM state. Since `document` methods do not touch application heap objects, Crom does not need to rewrite them.

The names of function parameters may shadow those of global variables. Speculative references to these variables should *not* resolve to `specContext`. When rewriting functions with shadowed globals, Crom passes a new speculative scope to `with()` statements; this scope is a copy of `specContext` that lacks references to shadowed globals.

If speculative code creates a new global property, it may slip past the `with` statement into the real global namespace. Fortunately, JavaScript is single-threaded, so Crom can check for new globals after the speculative handler has finished and sweep them into `specContext` before they are seen by other code.

When speculative code `deletes` a global property, this only removes the property from `specContext`. When the speculation commits, this property must also be deleted from the global namespace. To accomplish this, Crom rewrites `delete` statements to additionally collect a list of deleted property names. If the speculation later commits, Crom removes these properties from the real global namespace.

```
function genClosure(x){
   function f(){alert(x++);}
   return f;
}
var closureFunc = genClosure(0);
button0.onclick = closureFunc;
button1.onclick = closureFunc;
```

Figure 5: Variable `x` persists in a hidden closure scope.

```
function genClosure(x){
   var __cIndex = Crom.newClosureId();
   __closureEnvironment[__cIndex].x = x;
   function f(){
     alert(__closureEnvironment[__cIndex].x++);
   }
   f.__cIndex = __cIndex;
   return f;
}
```

Figure 6: The rewritten closure scope is explicit.

5.1.5 Externally Shared Closures

Whenever a function is created, it stores its lexical scope in an implicit activation record, using that scope for subsequent name resolution. Unfortunately, these activation records are not introspectable. If they escape cloning, they become state shared with the real (i.e., non-speculative) browser context. To avoid this fate, Crom rewrites closures to use explicit activation objects. Later, when Crom creates a speculative context, it can clone the activation object using its standard techniques.

Consider the code in Figure 5, which creates a closure and makes it the event handler for two different buttons. During non-speculative execution, clicking either button updates the same counter inside the shared closure. However, `closureFunc.toString()` merely returns "function (){alert(x++);}", with no mention of `x`'s closure binding. Using the rewriting techniques described so far, a rewritten handler would erroneously look for `x` in the global scope.

Figure 6 shows `genClosure()` rewritten to use an explicit activation record. `Crom.newClosureId()` creates an empty activation record, pushes it onto a global array `__closureEnvironments`, and returns its index. Crom rewrites each property that implicitly references the closure scope to explicitly reference the activation record via a function property `__cIndex`.

Later, when rewriting the `button0` or `button1` event handler, Crom detects that the handler is a closure by its `__cIndex` property. If the value of `f.__cIndex` is, say, 2, Crom rewrites the closure function by adding the statement `var __cIndex = 2;` immediately before the `with(specContext)` in the string passed to `eval()`. This gives the rewritten handler enough state to access the proper explicit activation record. Like any other global, `__closureEnvironments` is speculatively cloned, so each speculative execution has a private snapshot of the state of all closures.

Currently, applications must explicitly invoke Crom to rewrite functions which return closures. They do this by executing `g = Crom.rewriteCG(g)` for each closure-generating function `g`. Future versions of Crom will use lexical analysis to perform this rewriting automatically.

5.2 Committing Speculative State

Committing a speculative context is straightforward. First, Crom updates the DOM tree root in the non-speculative context, making it point to the DOM tree in the committing speculation's hidden `iframe`. Next, Crom updates the heap state. A `for-in` loop enumerates the heap roots in `specContext` and assigns them to the corresponding properties in the real global name space; this moves both updated and newly created roots into place. Finally, Crom iterates through the list of global properties deleted by the speculative execution and removes them from the real global name space.

5.3 AJAX Caching

To support the caching of AJAX results, client-side programs call the Crom methods `Crom.cacheAdd(key, AJAXresult)` and `Crom.cacheGet(key)`. These methods allow applications to associate AJAX results with arbitrary cache identifiers. Like the regular browser cache, Crom's AJAX cache is accessible to speculative and non-speculative code. For example, in the modified Decimail client, the event handler for the "fetch new message" operation is broken into two functions, `downloadMessage()` and `markMessageRead()`. `downloadMessage()` is read-only on the server side, but it modifies client-side state, e.g., a JavaScript array that contains metadata for each fetched message. Thus, when the Decimail client speculates on a message fetch, it rewrites `downloadMessage()`'s call tree and runs it in a speculative context. The speculative `downloadMessage()` looks for the message in Crom's cache and does not find it. It fetches the new email using AJAX and inserts it into Crom's cache. Later, when the user actually triggers the "fetch new message" handler, `downloadMessage()` runs in non-speculative mode and finds the requested email in the cache. Decimail then calls `markMessageRead()` to inform the server of the user's action.

5.4 Speculative Uploads

An upload form typically consists of a *file input* text box and an enclosing *submit form*. After the user types a file name into the text box, the input element generates an `onchange` event. However, the file is not uploaded to the server until the user triggers the `onsubmit` event of the enclosing form, typically by clicking a button inside the form. At this point, the application's `onsubmit` handler is called to validate the file name. Unless this handler returns `false`, the browser POSTs the form to the server and sends the server's HTTP response to the target of the form. By default, the target is the current window; this causes the browser to overwrite the current page with the server's response.

Crom implements speculative uploads with a client/server protocol. On the client, the developer specifies which file input should be made speculative by calling `Crom.prePost(fileInput,callback)`. Inside `prePost()`, Crom saves a reference to any user-specified `onsubmit` form-validation handler, since Crom will supply its own `onsubmit` handler shortly. Crom installs an `onchange` event handler for the file input which will be called when the user selects a file to upload. The handler creates a cloned version of the upload form in a new invisible `iframe`, with all file inputs removed except the one representing the file to speculatively upload. If the application's original `onsubmit` validator succeeds, Crom's `onchange` handler POSTs the speculative form to a server URL that only accepts speculative file uploads. Crom's server component caches the uploaded file and its name, and the client component records that the upload succeeded.

`Crom.prePost()` also installs an `onsubmit` handler that lets Crom introspect the form before a real click would POST it. If Crom finds a file that has already been cached at the server, Crom replaces the associated file input with an ordinary text input having the value `ALREADY_SENT:`*filename*. Upon receipt, Crom's server component inserts the cached file data before passing the form to the application's server-side component.

The interface given above is least invasive to the application, but a speculation-aware application can provide upload progress feedback to the user by registering a progress callback with Crom. Crom invokes this handler in the real domain when the speculative upload completes, allowing the application to update its GUI.

5.5 Optimizations

To conclude this section, we describe three techniques for reducing speculative cloning overheads.

5.5.1 Lazy Cloning

For complex web sites, *eager cloning* of the entire application heap may be unacceptably slow. Thus, Crom offers a *lazy cloning* mode in which objects are only copied when a speculative execution is about to access them. Since this set of objects is typically much smaller than the set of all heap objects, lazy cloning can produce significant savings.

In lazy mode, Crom initially copies only the DOM tree and the heap variables referenced by DOM nodes. As the speculative computation proceeds, Crom dynamically rewrites functions as before. However, object cloning is now performed as a side effect of the rewriting process. Crom's lexical analysis identifies which variable names refer to locals, globals, and function parameters. Locals do not need cloning. Strictly speaking, Crom only needs to clone globals that are written by a speculative execution; reads can be satisfied by the non-speculative objects. However, a global that is read at one point in a call chain may be passed between functions as a parameter and later written. To avoid the bookkeeping needed to track these flows, Crom sacrifices performance for implementation simplicity and clones a global variable whenever a function reads or writes it. The function is then rewritten using the techniques already described. Function parameters need not be cloned as such—if they represent globals, they will be cloned by the ancestor in the call chain that first referenced them.

Lazy cloning may introduce problems at commit time. Suppose that global objects named `X` and `Y` have properties `X.P` and `Y.P` that refer to the same underlying object `obj`. If a speculative call chain writes to `X.P`, Crom will deep-copy `X`, cloning `obj` via `X.P`. The speculation will write to the new clone `obj'`. If the call chain never accesses `Y`, `Y` will not be cloned since it is not reachable from the object tree rooted at `X`. Later, if the speculation commits, Crom will set the real global `X` to `specContext.X`, ensuring that the real `X.P` points to `obj'`. However, `Y.P` will refer to the original (and now stale) `obj`.

In practice, we have found that such stale references arise infrequently in well-designed, modular code. For example, the autocompletion widget is a stand-alone piece of JavaScript code. When a developer inserts a speculative version of it into an enclosing web page, the widget will not be referenced by other heap variables, and running in lazy mode as described will not cause stale child references. Regardless, to guarantee correctness at commit time, Crom provides a *checked lazy mode*. Before Crom issues any speculative computations, it traverses every JavaScript object reachable from the heap roots or the DOM tree, and annotates each object with `__parents`, a list of parent pointers. A parent pointer identifies the parent object and the property name by which the parent references the child. Crom copies `__parents` by reference when cloning an object. When a lazy speculation commits, Crom uses the `__parents` list to update stale child references in the original heap.

Crom also has an *unchecked lazy mode* in which Crom clones lazily but assumes that stale child references never occur, thereby avoiding the construction and the checking of the parent map. For applications with an extremely large number of objects and/or a highly convoluted object graph, unchecked lazy mode may be the only cloning technique that provides adequate performance. We return to this issue in the evaluation section. For now, we make three observations. First, our experience has been that the majority of event handlers will run safely in unchecked mode without modification. Second, Crom provides an interactive execution mode that allows developers to explicitly verify whether their speculative event handlers are safe to run in unchecked lazy mode. During speculative commits in interactive mode, Crom reports which committing objects have parents that were not lazily cloned and thus would point to stale children post-commit. Crom automatically determines the object tree roots that the programmer must explicitly reference in the event handler to ensure that the appropriate objects are cloned. The programmer can then perform this simple refactoring to make the handler safe to run in unchecked lazy mode.

Third, and most importantly, Section 6 shows that most of Crom's benefits arise from speculatively warming the browser cache. Committing speculative DOM nodes and heap objects can mask some computational latency, but network fetch penalties are often much worse. Thus, if speculating in checked lazy mode is too slow, or checked mode refactoring is too painful, an application can pass a flag (not shown in Figure 1) to `Crom.makeSpeculative()` which instructs Crom to discard speculative contexts after the associated executions have terminated. In this manner, applications can use unchecked lazy speculations solely to warm the browser cache, forgoing complications due to commit issues, but deriving most of the speculation benefit.

5.5.2 Speculation Zones

An event handler typically modifies a small fraction of the total application heap. Similarly, it often touches a small fraction of the total DOM tree. Lazy cloning exploits the first observation, and *speculation zones* exploit the second. An application may provide an optional `DOMsubtree` parameter to `Crom.makeSpeculative()` that specifies the root of the DOM subtree that an event handler modifies. At speculation time, Crom will only clone the DOM nodes associated with this branch of the tree. At commit time, Crom will splice in the speculative DOM branch but leave the rest of the DOM tree undisturbed.

Speculation zones are useful when an event handler is tightly bound to a particular part of the visual display. For example, the autocompletion widget in Figure 3 only modifies the `<div>` tag that will hold the fetched search results. Speculation zones are also useful when the DOM tree contains rich objects whose internal state is opaque to the native `cloneNode()` method. Examples of such objects are Flash movies and Java applets. When `cloneNode()` is invoked upon such an object, the returned clone is "reset" to the initial state of the source object; for example, a Flash movie is rewound to its first frame. Since it is difficult for JavaScript code to reason about the state of such objects, applications can use speculation zones to "speculate around" these opaque parts of the DOM tree.

5.5.3 Context Pools

Crom hides speculative computations within the "think time" of a user. The shorter the think time, the faster Crom must launch speculations to reduce user-perceived fetch latencies. For example, time pressure is comparatively high for the autocompletion widget (§4.1.1) since a fast typist may generate her search string quickly.

To reduce the synchronous overhead of issuing new speculations, applications can call `Crom.createContextPool(N, stateSketch, DOMsubtree)`. This method generates and caches `N` clones of the current browser environment, tagging each with the sketch value generated when `stateSketch` is passed the current browser context.

Using context pools, the entire cost of eager cloning or the initial cost of lazy cloning is paid in advance. At speculation time, Crom finds an unused context that is tagged with the application's current sketch. Crom immediately specializes it using a mutator and issues the new speculation.

5.6 Limitations and Future Work

Many of the limitations of the current Crom prototype arise from the fact that the client-side portion runs as slow JavaScript code instead of fast C++ code inside the browser. For example, we pursued the optimizations described in Section 5.5 after discovering that in many cases, copying the entire browser context using JavaScript code would result in unacceptably slow performance. Ideally, browsers would natively support context cloning, and applications could always use checked mode speculations without fear of excessive CPU usage.

Speculative fetches compete with non-speculative fetches for bandwidth. A native implementation of Crom could measure the traffic across multiple flows and prioritize non-speculative fetches over speculative ones. Our JavaScript implementation cannot measure such browser-wide network statistics. However, developers can use `Crom.maxSpeculations(N)` to place an upper limit on the number of speculations that can be triggered by a single call to `Crom.forceSpeculations()`.

As mentioned in Section 5.5.2, opaque browser objects like applets and Flash applications cannot be introspected by JavaScript code. Our JavaScript Crom library can only clone them in a crude fashion by recreating their HTML tags (and thereby reinitializing the clones to a virgin state). In practice, we expect most applications to avoid this issue by using speculation zones. However, some applications might benefit from the ability to clone rich objects in more sophisticated ways. An in-browser implementation of Crom could do this, but we leave an exploration of these mechanisms for future work.

The parsing engine for our client-side rewriter is fairly unsophisticated. Implemented with regular expressions, it is sufficient for analyzing the real applications described in the next section. However, it does not parse the complete JavaScript grammar. Future versions of the client-side library will use an ANTLR-driven parser [18]. An in-browser implementation could obviously reuse the browser's native parsing infrastructure.

6 Evaluation

Ideally, we would evaluate Crom by taking a preexisting application that has custom speculation code, replacing that code with Crom calls, and comparing the performance of the two versions. Unfortunately, custom speculation code is often tightly integrated with the rest of the application's code base, making it difficult to remove the code in a principled way and provide a fair comparison with Crom's speculation API. Thus, our evaluation explores the performance of the real applications from Section 4 that we modified to use the Crom API. We show that Crom's computational overheads are hideable within user think time, and that Crom can reduce user-perceived fetch latencies by an order of magnitude under realistic network conditions.

Our Crom prototype has been tested most extensively on the Firefox 3.5 browser. Its core functionality has also been tested on IE7 and Safari 4.0. However, we are still fixing compatibility issues with the latter set of browsers, so this section only contains Firefox results.

6.1 Performance of Modified Applications

To test the speculative autocompletion widget and tab manager, we downloaded real web pages onto our local web server; Figure 7 lists the pages that we examined. We inserted the Crom library and the speculative applications into the pages, then loaded the pages using a browser to test their performance. Decimail is a standalone application, so we tested it by itself, i.e., we did not

Web site	Primitives	Objects	Functions	DOM nodes
Google	16	5	31	77
Gmail-o	2574	622	6	66
Gmail-i	0	0	3	1999
ESPN	1053	414	596	1422
YouTube	329	119	894	823
Live	49	8	80	184
Yahoo	150	26	51	694
MySpace	1447	236	904	499
eBay	826	589	1360	633
MSN	227	87	210	932
Amazon	13112	2895	1145	3003
CNN	859	129	588	1733

Figure 7: The types of JavaScript variables in several popular web pages. Gmail-o refers to Gmail's outer control frame and Gmail-i refers to the inner frame which displays the inbox message list.

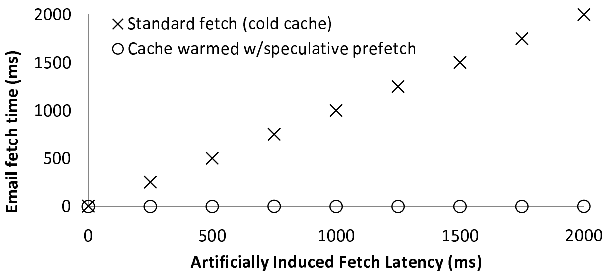

Figure 8: Latency reductions using Crom's AJAX cache.

embed it within an enclosing web page. Stripped of comments and extraneous whitespace, Crom's JavaScript code added 65 KB to each application's download size. All experiments ran on an HP xw4600 workstation with a dual-core 3GHz CPU and 4 GB of RAM. Web content was fetched from a custom localhost web server that introduced tunable fetch delays. This allowed us to measure Crom's latency-hiding benefits as a function of the fetch penalty. All experiments represent the average of 10 trials. Standard deviations were less than 6% in all cases.

6.1.1 Decimail Client

The modified Decimail client [5] used Crom's AJAX cache to store results from speculative mail fetches. Running in unchecked lazy mode, Decimail's cloning and rewriting overheads were less than 5 ms per fetch, so we elide further discussion of them. Figure 8 shows the benefit of finding a requested message in the Crom cache instead of having to fetch it synchronously. Unsurprisingly, a cache hit took no more than 3 ms to serve, whereas the cache miss penalty was the fetch latency to the server.

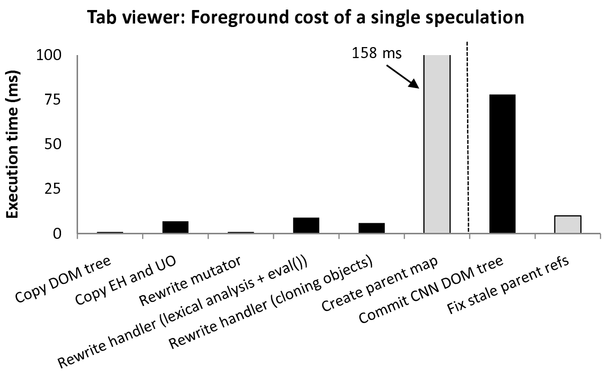

Figure 9: Tab manager pre- and post-commit overheads (checked mode costs in grey).

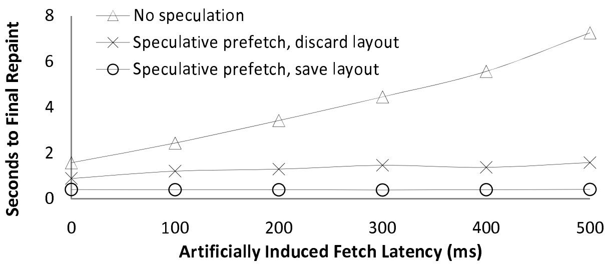

Figure 10: User-perceived latencies for tab manager.

6.1.2 Tab Manager

Figure 9 depicts the speculative overheads in the tab manager application [12]. In this experiment, the tab manager was embedded within the `ESPN` front page. Clicking the "make new tab" button generated an AJAX request for a web page. Once fetched, the page was rendered inside a `<div>` tag; this tag was a child of the enclosing `<div>` tree controlled by the tab manager. Crom used this enclosing tag as the speculation zone. We configured the tab manager to speculatively fetch the `CNN` home page, a complex page with over 1700 DOM nodes.

Figure 9 breaks the speculation overheads into pre-commit costs (the left side of the dotted line) and during-commit costs (the right side of the line). The black bars represent the overheads for unchecked lazy mode. The grey bars represent the additional costs that would arise from running lazy cloning in checked mode.

Unchecked lazy mode: When speculating in unchecked lazy mode, aggregate pre-commit speculation costs were low, totalling 24 ms. Copying the speculated-upon DOM tree was fast (1 ms), since the speculation zone consisted of a small `<div>` tree of depth 3. Walking the cloned DOM tree and copying event handlers and user-defined objects took only 7 ms. There was no mutator to rewrite, but Crom did have to rewrite an event handler call tree with a maximum depth of 4. Figure 9 breaks the rewriting cost into two parts.

The first part represents the cost of lexical analysis, source code modification, and calling `eval()` on the modified source. The second part represents the cost of copying heap objects during the rewriting process. Both costs were less than 10 ms each since the call chain touched a minority of the total page state.

At commit time, there were no stale child references to fix since Crom was running in unchecked mode. Committing speculative heap objects took less than 1 ms, since Crom merely had to make global variables in the real domain point to speculative object trees. Over 99% of the commit overhead was generated by the splice of the speculative DOM subtree into the non-speculative one. Although the splice was handled by native code, it required a screen redraw, since the formerly invisible `CNN` context was now visible. Screen redraws are one of the most computationally intensive browser activities. However, Crom avoided the additional (and more expensive) *reflow* cost, since the layout for the `CNN` content was determined during the speculative execution. Since even non-speculative computations must pay the redraw cost upon updating the screen, Crom added less than 1 ms to the inherent cost of displaying the new content.

Checked lazy mode: In this mode, Crom built a parent map before issuing any speculations and checked for stale object references at commit time. The grey bars in Figure 9 depict these costs, which must be paid in addition to the rewriting and cloning costs in black. Constructing the parent mapping took 158 ms, leading to an overall pre-commit cost of 182 ms. Although the mapping cost is amortizable across multiple speculations, an aggregate CPU overhead of 182 ms pushes against the limits of acceptability. The current implementation of Crom does not stage its pre-commit operations, i.e., Crom holds the processor for the entire duration of the cloning, rewriting, and parent mapping process. Depending on the application, a pre-commit overhead of 182 ms may interfere with foreground, non-speculative JavaScript that needs CPU cycles. Recent versions of Firefox provide a JavaScript `yield` statement for implementing generators; future versions of Crom may be able to stage pre-commit costs using such generators.

When committing a checked-mode speculation, Crom must patch stale object references. In this experiment, the cost was only 5 ms. As explained in Section 6.2.4, the patching overhead is proportional to the number of objects cloned, which is typically much smaller than the total number of objects in the application.

User-perceived benefits: To keep the GUI responsive, long redraws in Firefox are split into a synchronous part and several asynchronous ones. The commit-time black bar in Figure 9 only captures the synchronous cost. Figure 10 quantifies the end-to-end *user-perceived* latency reduction that is enabled by speculative prefetching and pre-layout. The injected web server latency varies on the x-axis, and the y-axis provides the delay between the start of the page load and the final screen repaint; repaint activity was detected using the custom Firefox event `MozAfterPaint`.

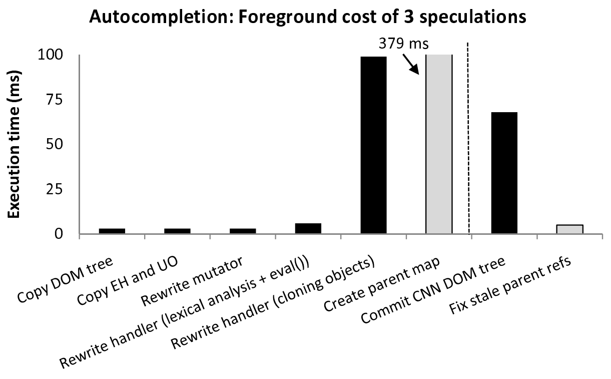

Figure 11: Autocompleter pre- and post-commit costs (checked-mode costs in grey).

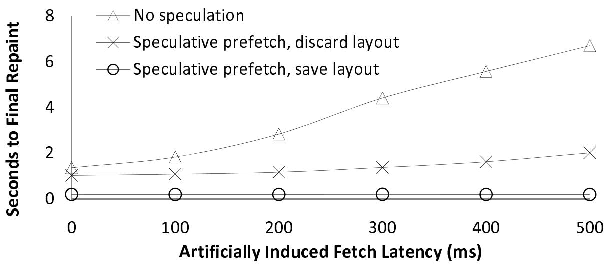

Figure 12: User-perceived latencies for autocompleter.

Figure 10 shows that speculative execution can dramatically reduce user-perceived latencies. For example, with a 300 ms fetch penalty, Crom needed 399 ms to complete the page load, whereas a non-speculative synchronous load with a cold cache required 3,427 ms. Speculatively prefetching the data but discarding the layout (i.e., not committing the speculative context) required 1,302 ms of user-perceived load time. The load penalty relative to the full speculative case arose from two sources: recalculating the layout, and for some objects, waiting for the server to respond to cache validation messages. Such messages are not generated when a precomputed DOM subtree is inserted into the foreground DOM tree.

6.1.3 Autocompletion Widget

To evaluate the speculation overheads for the autocompleter [21], we embedded it inside the `Amazon` front page, simulating a new search box for the page. When the user hit the widget's "submit" button, the widget used AJAX to fetch and display content associated with the user's search terms. The autocompleter speculated after the user had typed two letters, and we limited the number of speculations to three. To provide a comparison with the tab manager experiment, the widget always fetched the CNN front page. Crom ran in lazy mode with a speculation zone consisting of the <div> tag that would display the fetched content.

Figure 11 depicts the speculation overheads. Examining the overheads from left to right, we see that copying the DOM tree, the event handlers, and the user objects was extremely fast. This is because the autocompleter used a speculation zone that limited the DOM copying to a single div subtree. The mutator function was also very simple, so rewriting it was cheap. However, rewriting the event handler call tree was not cheap due to high cloning costs. The JavaScript objects that implemented the autocompletion logic were fairly complex, and copying the whole set took 33 ms. Since the autocompleter issued three speculations, this object set had to be copied three times.

In unchecked lazy mode, the total pre-commit speculation cost was 114 ms, not counting the 22 ms needed to preemptively create three speculative domains (see Section 5.5.3 for details). Creating a parent mapping would add 379 ms, making the total pre-commit overhead a prohibitive 493 ms. Thus, the autocompleter requires unchecked lazy mode to make Crom's speculations feasible.

Commit costs were similar for the tab manager and the autocompletion widget. As Figure 12 shows, the reduction in user-perceived latency was equally dramatic.

6.2 Exploring Speculation Costs

The previous section showed that Crom's overheads are low when using unchecked lazy cloning and speculation zones. In this section, we examine the costs of full heap and checked lazy speculation, as well as the cost of copying the entire DOM tree. These types of speculation require the least effort from the web developer, since there is no need to identify speculable DOM subtrees or perform checked-mode refactoring. We believe that these activities are fairly straightforward for well-designed code. Unfortunately, poorly written JavaScript code abounds, and even good developers may generate tangled code when working on a constantly evolving web page. Thus, it is important to understand when checked mode or full copy speculation is feasible when the Crom API is not natively implemented by the browser.

6.2.1 Copying the Full Application Heap

Figure 7 describes the JavaScript environment for several popular web sites[2]. As expected, there is wide variation across sites. Figure 13 shows the time needed by

[2]Figure 7 provides a lower bound on the number of variables in each site; Crom cannot discover objects which are only reachable by object trees rooted in DOM 2 handler state.

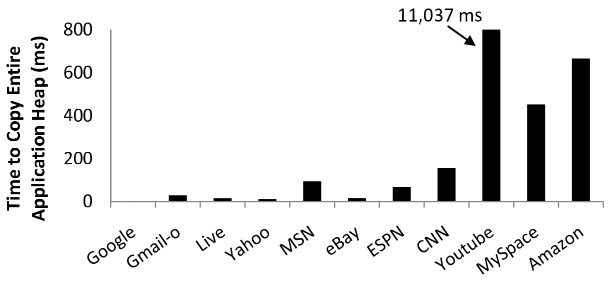

Figure 13: Copying the full application heap.

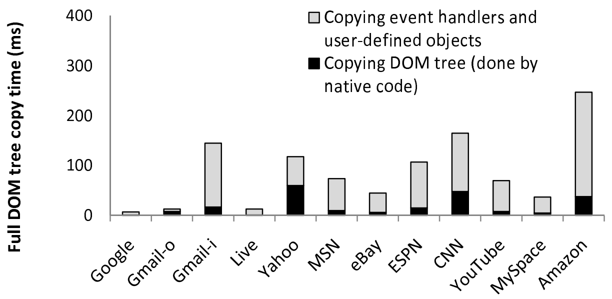

Figure 14: Copying the full DOM tree.

Firefox to copy the full application heap in these sites. For many sites, the one-time copy overhead was low. For example, in a fairly complex site like MSN, the copy cost was 94 ms. However, this is a *per-speculation* cost, since each speculative execution needs a private copy of the heap. The larger the cost, the fewer speculations can be issued in a window of a few hundred milliseconds.

For more complex sites like Amazon, YouTube, and MySpace, full heap copying was prohibitively expensive, mainly due to the overhead of function cloning. This overhead was proportional to both the number and the complexity of the functions. So, even though MySpace and YouTube had a similar number of functions (904 versus 894), it took 452 ms to clone MySpace's heap, but 11 *seconds* to clone YouTube's heap. The high cloning overhead for YouTube is a consequence of its sophisticated JavaScript functionality: it contains over 200K of code for manipulating Flash movies, performing click analytics, and managing advertisements. For sites like this, full heap copying is completely infeasible, making lazy cloning a prerequisite for high-performance speculation.

6.2.2 Copying the Full DOM Tree

To copy the DOM tree (or a branch of the tree), Crom first calls `cloneNode(true)` on the root of the tree. Then, Crom traverses the cloned tree and the base tree in parallel, reference-copying the event handlers and deep-copying the user-defined objects belonging to the non-speculative DOM nodes. Figure 14 shows the relative costs of these two steps. Since `cloneNode()` is imple-

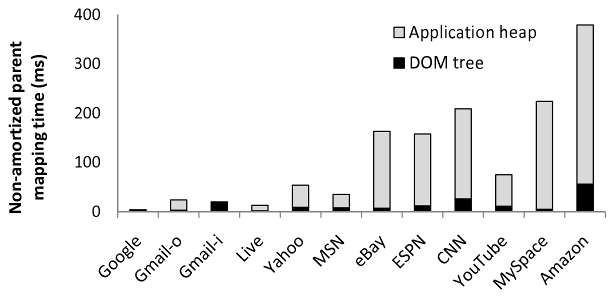

Figure 15: Creating a parent map.

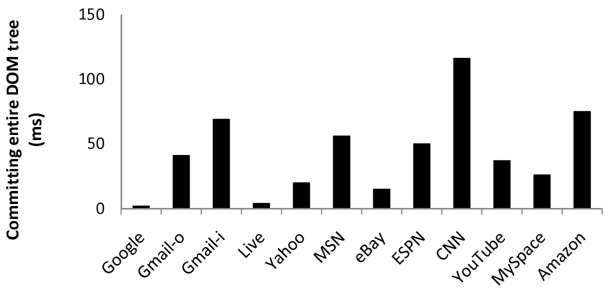

Figure 16: Committing a full DOM tree.

mented in native code, the first step was very fast. The second step was more onerous since it was implemented by non-native Crom code. The aggregate copy cost represents a per-speculation overhead, since each speculative computation must possess a private copy of the tree. Amazon had the highest copy overhead (247 ms), but 40% of the remaining websites had copy penalties above 100 ms. Thus, we expect that most speculative web sites will avoid full DOM copying and use speculation zones.

6.2.3 Creating the Parent Map

Figure 15 shows the cost of creating a parent map in various web sites. The overhead is split into two parts: the cost of traversing the DOM tree, and the cost of walking the object forest rooted in the application heap. Traversing the DOM tree was much faster than traversing the application heap. Whereas Crom could get a list of all DOM nodes using the native code `document.getElementByTagName("*")`, it had to walk the application heap using a user-level breadth-first traversal.

Creating a parent map and updating stale references is unnecessary if Crom copies the entire application heap. However, given the choice between unchecked execution with full heap copying, and checked execution with lazy copying, many applications will prefer the latter. This is because the overhead of parent mapping is amortizable across multiple speculations, whereas full heap copy costs cannot be shared. Comparing Figures 13 and 15 reveals that for sites with small enough heaps to make full heap copying reasonable, creating the parent map is often no more expensive than a single full heap copy; thus, when issuing multiple speculations, lazy copying quickly repays the initial parent mapping overhead.

6.2.4 Committing a Speculative Domain

Committing a speculative context requires three actions. First, the speculative DOM tree must be spliced into the real DOM tree. Second, the cloned object trees from the application heap must be inserted into the real global name space. Third, if Crom is running in checked mode, the non-speculative parents of speculative objects must have their child references patched.

Repairing the heap name space is essentially free—for each reference to an object tree root, Crom just makes it point to the speculative copy of the root object. The time required for parent patching is $O(SP)$, where S is the number of speculatively cloned objects and P is the average number of parents per object. S is typically much smaller than the total number of objects, so in our experience, fixing stale child references takes no more than 20 ms.

Splicing the speculative DOM tree into the real one is handled by native code. However, the splice may be the slowest step of the commit if the speculative tree is large. In these situations, the browser is forced to re-render large regions of the display, a computationally intensive task. Figure 16 shows the splice costs in the absence of speculation zones, i.e., when the entire DOM tree is copied and then reinserted. With the exception of `CNN`, each splice took less than 75 ms. Regardless, DOM splicing is a cost that must be paid by both speculative and non-speculative code, and even checked commits add little overhead to the fundamental cost of updating the browser display.

7 Related Work

File systems often use prefetching to hide disk and network latency [13, 19], and many of these techniques can be applied to the web domain. Empirical studies show that over half of all web objects can be effectively prefetched, doubling the latency reductions available from caching alone [14]. Various academic and commercial projects have tried to exploit this opportunity [4, 15, 20]. For example, Padmanabhan's algorithm uses server-collected access statistics to generate prefetching hints for clients [17]. The current draft of the HTML 5 protocol allows such hints to be specified using a new "prefetch" attribute for `<link>` tags [23]. The Fasterfox extension for Firefox automatically prefetches statically referenced content in a page [10]. All of these approaches are limited to prefetching across static content graphs. In contrast, Crom allows prefetching in web pages with interactive client-side code and dynamic content.

Speculative execution has been used to drive prefetching in file systems [1, 8] and parallel computation systems [2]. In these environments, a process is speculated forward, possibly on incorrect data, to discover what I/O requests may appear in the future. The sole purpose of speculative execution is to warm the cache; other side effects are discarded. In contrast, Crom speculates on user activity rather than the results of I/O operations. When appropriate, Crom can also commit speculative contexts to hide computational latencies associated with screen redraws.

The Speculator Linux kernel [16] supports speculative execution in distributed file systems. A client-side file system process speculates on the results of remote operations and continues to execute until it tries to generate an externally visible output like a network packet. The process is blocked until its speculative input dependencies are definitively resolved. At that point, the process is either marked as non-speculative or rolled back to a checkpoint. Unlike Speculator, Crom has a server-side component which allows speculations to externalize network activity. However, Crom requires applications to be modified, whereas Speculator only modifies the OS kernel.

8 Conclusion

In this paper, we describe why speculative execution is a natural optimization technique for rich web applications. We introduce a high-level API through which applications can express speculative intent, and a JavaScript implementation of this API that runs on unmodified browsers. This implementation, called Crom, automatically converts event handlers to speculative versions and runs them in isolated browser contexts, caching both the fetched data and the screen layout for that data. Experiments show that Crom-enabled applications can reduce user-perceived delays by an order of magnitude, greatly improving the browsing experience. By abstracting away the programmatic details of speculative execution, Crom successfully lowers the barrier to producing rich, low-latency web applications.

References

[1] Chang, F., and Gibson, G. A. Automatic I/O hint generation through speculative execution. In *Proc. 15th ACM Symposium on Operating System Design and Implementation (OSDI)* (1999), pp. 1–14.

[2] Chen, Y., Byna, S., Sun, X.-H., Thakur, R., and Gropp, W. Hiding I/O latency with pre-execution prefetching for parallel applications. In *Proc. ACM/IEEE Conference on Supercomputing (SC)* (2008), pp. 1–10.

[3] Crovella, M., and Barford, P. The network effects of prefetching. In *Proc. IEEE Infocom* (1998), pp. 1232–1239.

[4] Davison, B. D. Assertion: Prefetching with GET is not good. In *Proc. 6th International Workshop on Web Caching and Content Distribution (WCW)* (2001), pp. 203–215.

[5] Decimail project. `http://freshmeat.net/projects/decimail/`.

[6] Fielding, R., Gettys, J., Mogul, J.,Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999.

[7] Flanagan, D. *JavaScript: The Definitive Guide, Fifth Edition*. O'Reilly Media, Inc., Sebastopol, CA, 2006.

[8] Fraser, K., and Chang, F. Operating system I/O speculation: How two invocations are faster than one. In *Proc. USENIX Annual Technical Conference* (2003), pp. 325–338.

[9] Garrett, J. Ajax: A New Approach to Web Applications. `http://www.adaptivepath.com/ideas/essays/archives/000385.php`, February 18, 2005.

[10] Gentilcore, T. Fasterfox Firefox Extension. `http://fasterfox.mozdev.org`, 2005.

[11] Jones, R., and Lins, R. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[12] Kalleland, A. DHTML tab widget. `http://www.dhtmlgoodies.com/scripts/tab-view/tab-view.html`.

[13] Korner, K. Intelligent caching for remote file service. In *Proc. International Conference on Distributed Computing Systems (ICDCS)* (May/Jun. 1990), pp. 220–226.

[14] Kroeger, T. M., Long, D. D. E., and Mogul, J. C. Exploring the bounds of web latency reduction from caching and prefetching. In *Proc. 1st USENIX Symposium on Internet Technologies and Systems (USITS)* (1997), pp. 2–2.

[15] Markatos, E. P., and Chronaki, C. E. A top-10 approach to prefetching on the Web. In *Proc. 8th Annual Conference of the Internet Society (INET)* (1998).

[16] Nightingale, E. B., Chen, P. M., and Flinn, J. Speculative execution in a distributed file system. In *Proc. 20th ACM Symposium on Operating Systems Principles (SOSP)* (2005), pp. 191–205.

[17] Padmanabhan, V. N., and Mogul, J. C. Using predictive prefetching to improve World Wide Web latency. *Computer Communication Review 26* (1996), 22–36.

[18] Parr, T. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[19] Patterson, R. H., Gibson, G. A., Ginting, E., Stodolsky, D., and Zelenka, J. Informed prefetching and caching. In *Proc. 15th ACM Symposium on Operating Systems Principles (SOSP)* (1995), pp. 79–95.

[20] Pitkow, J., and Pirolli, P. Mining longest repeating subsequences to predict World Wide Web surfing. In *Proc. 2nd USENIX Symposium on Internet Technologies and Systems (USITS)* (1999), pp. 13–13.

[21] script.aculo.us JavaScript Library. `http://script.aculo.us/`.

[22] Stockwell, C. IE8 Performance. `http://blogs.msdn.com/ie/archive/2008/08/26/ie8-performance.aspx`.

[23] Web Hypertext Application Technology Working Group. HTML 5 Draft Standard. `http://whatwg.org/html5`, fetched July 2, 2009.

[24] World Wide Web Consortium. Document Object Model (DOM). `http://www.w3.org/DOM/`, 2005.