ZOOMM: A Parallel Web Browser Engine for Multicore Mobile Devices

Călin Caşcaval Seth Fowler Pablo Montesinos Ortego Wayne Piekarski Mehrdad Reshadi
Behnam Robatmili Michael Weber Vrajesh Bhavsar
Qualcomm Research Silicon Valley
zoomm@qualcomm.com

Abstract

We explore the challenges in expressing and managing concurrency in browsers on mobile devices. Browsers are complex applications that implement multiple standards, need to support legacy behavior, and are highly dynamic and interactive. We present ZOOMM, a highly concurrent web browser engine prototype and show how concurrency is effectively exploited at different levels: speed up computation performance, preload network resources, and preprocess resources outside the critical path of page loading. On a dual-core Android mobile device we demonstrate that ZOOMM is two times faster than the native WebKit based browser when loading the set of pages defined in the Vellamo benchmark.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming - Parallel Programming

Keywords Parallel Browser, Mobile Multicore

1. Introduction

Smartphones and tablet computers are seeing tremendous adoption by consumers. These devices are replacing laptops and desktop machines as the platform of choice for many users. Mobility and permanent connectivity have driven initial adoption and as the devices become more capable, most users can accomplish their daily tasks with ease and convenience. Many of these tasks involve web browsing, whether using a browser or native applications that provide a customized view of web content.

In this paper we present a browser architecture targeted toward exploiting the hardware capabilities of modern mobile devices: multicore parallelism and hardware acceleration, increased network bandwidth and long network latencies. The majority of current smartphones and tablets have SoCs with 2 or 4 cores, aggressively optimized for power – power gated, voltage and frequency scaled. On the network side, LTE brings 100 Mbps bandwidth, however, latency continues to be high [1].

Web browser designers have to address several challenges: fast response time for page load, even in the presence of long network latencies [22], high performance to enable interactivity for web applications, and user interface responsiveness to provide good experience [18]. In this paper we demonstrate how our browser architecture allows exploitation of multicore concurrency to hide network latency and improve performance. When tested using the publicly available Vellamo [20] benchmark suite, our ZOOMM browser completed the run in 55 seconds, compared to 113 seconds using the standard WebKit [23] available on a commercial HTC Jetstream device. This is an approximately 2× improvement in performance, demonstrating the benefits of our browser architecture.

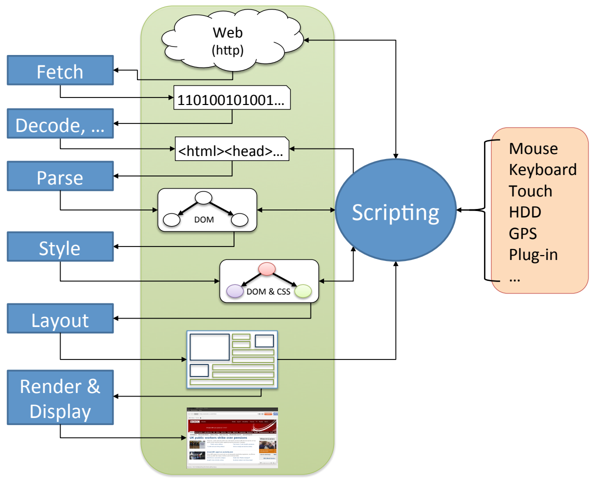

Figure 1. Typical browser processing steps.

Exploiting concurrency to improve browser experience is a relatively new approach. Most existing browsers, such as the WebKit based Chrome [4] and Safari [19], along with the Firefox [8] browser, have a long history of development and are fundamentally architected as sequential engines, using event driven models to help with interactivity. Such design admits some limited parallelization; however, full parallelism requires thread safe data structures and synchronization between components that is hard to graft on an existing design. These browsers have been exploiting process multicore concurrency, using on process per tab, and relying on the OS to map processes to different cores.

As the Web is evolving, we see a remarkable increase in complexity and dynamic behavior. For example, in [13] the authors measured WebKit execution and observed that JavaScript took around 5% of the execution time. About one year later, the fraction of JavaScript execution has increased to 30%. Even more significantly, we observe a major trend to support application development using web technologies, such as HTML5, CSS, and JavaScript. Poor browser performance is one of the most important factors hampering the move towards web apps.

PPoPP'13, February 23–27, 2013, Shenzhen, China.
Copyright © 2013 ACM 978-1-4503-1922-5/13/02. . . $15.00

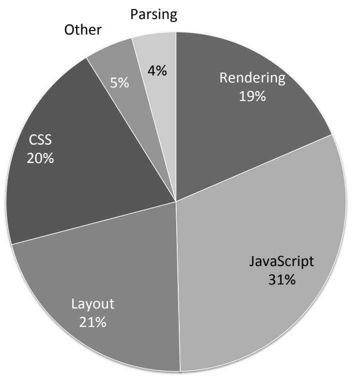

Figure 2. Browser processing times by component, excluding network load time. Profiling results obtained using the WebKit browser on an ARM Cortex A9 processor. Results are an aggregate of the top Alexa 30 sites as of March 2010.

Therefore our main challenge, and the focus of this paper, is to build a high-performance modern browser engine that works for a realistic set of web pages and web apps. In this paper we make the following contributions:

- A parallel browser architecture designed for fast web page loading and performant web applications; our hierarchical architecture exploits concurrency by overlapping the execution of major components as well as implementing parallel algorithms for some of the processing steps (Section 2);
- We demonstrate the use of concurrency to hide latency by discovering resources ahead of time, downloading and processing them in parallel (Section 3);
- We present a novel CSS matching and styling algorithm that scales linearly (Section 4);
- We describe briefly a parallel JavaScript engine to improve the performance of long running web applications (Section 5).

We briefly describe the other browser components and their interaction in Section 6 and discuss performance results in Section 8.

2. A Parallel Browser Architecture

Figure 1 shows a typical browser architecture and illustrates the steps required to render a web page. JavaScript interacts with the page during the page load, as well as after the page is loaded to provide interactivity. Figure 2 shows the breakdown of execution time by component, excluding the network time. Our measurements, similar to [22] show the network time being 30%-50% of the total execution time. Given this breakdown of computation, it is clear that in order to optimize the execution of the browser, one has to address all components.

2.1 Design goals

Our goal is to exploit concurrency at multiple levels: parallel algorithms for individual passes to speed up processing of each component, and overlapping of passes to speed up total execution time. In addition, we must respect the HTML and JavaScript semantics, even during concurrent execution. The main data structure that is used by all browser passes is the *Document Object Model* (DOM). The DOM is a tree representing all the HTML elements: their contents, relationships, styles, and positions. Web programmers use JavaScript to manipulate the DOM, producing interactive web pages and web apps. Most communication between browser passes and components happens through the DOM. Unfortunately, even in a concurrent browser, access to the DOM tree (constructed by the *HTML5 parser*) must be serialized, to conform to the HTML5 specification [11]. This is the biggest limitation ZOOMM must contend with, and it has influenced the design significantly. In our architecture, we manage access to the DOM through a dispatcher; most passes have their own private concurrent data structures to allow for greater parallelism inside components, and they send DOM updates to be processed by the dispatcher. Figure 3 shows the high level components of the architecture. We discuss the details in the following sections.

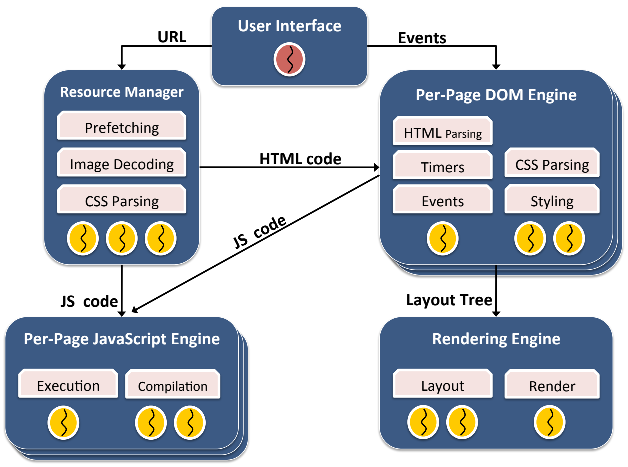

Figure 3. ZOOMM Browser Architecture. Concurrency is exploited both across components and within each component.

2.2 ZOOMM Browser Architecture

The ZOOMM browser consists of a number of loosely-coupled subsystems, which are all designed with concurrency in mind. With the exception of the browser-global *resource manager* and the *rendering engine*, all sub-systems are instantiated once for each page (shown as a separate *tab* in the user interface).

Resource Manager. The *resource manager* is responsible for managing and preprocessing all network resources, including fetching resources from the network, cache management for fetched resources, and notifications for the arrival of data from the network to other browser components. In our first implementation, all resources are fetched in the order in which they appear, without imposing any priorities. In addition, the *resource manager* includes other components, such as the *HTML prescanner* and *image decoder*. The HTML prescanner quickly determines all external resources in an HTML document, requests their downloading, and, depending on the type of resources, request further processing (Section 3.1). The Image decoder component consists of a thread pool that decodes images for later use as they are received by the resource manager. These operations are fully concurrent, as each image decode is an independent task.

DOM Engine. In ZOOMM, each page (tab) instantiates a *DOM engine* which consists of the *DOM dispatcher*, *HTML parser*, *CSS parsing and styling*, and *timers and events*. The *DOM dispatcher* thread is responsible for scheduling DOM updates; it serves as the page event loop. It serializes access to the DOM and manages the interaction between components. The rest of the browser infrastructure dispatches *work items* into the concurrent DOM dispatcher queue, which are then handled one at a time. Work items represent browser passes as well as events like from timers events and the user interface. The *HTML parser* receives incoming (partial) data chunks for an HTML document via a DOM dispatcher work item, and constructs the *DOM tree* by executing the HTML5 parsing algorithm [11]. The parser adds external resources referenced from the HTML document to the resource manager's fetch queue. The parser also initiates execution of JavaScript code by calling the *JavaScript engine* at appropriate times during parsing. The CSS engine is responsible for calculating the look and feel of the DOM elements for the later layout and rendering stages. Similar to image decoding, the resource manager hands off CSS stylesheets to the *CSS engine* for parsing and for discovering new resources to be requested (Section 4).

Rendering engine. Whenever the DOM or the CSS stylesheets change, whether because the fetcher delivered new resources, the HTML parser updated the DOM, or as a result of JavaScript computations, this change needs to be reflected on the screen so that the user can view and interact with it. The *layout engine* is responsible for transforming the styled DOM tree into geometry and content which the *rendering engine* can turn into a bitmap (Section 6). Ultimately this bitmap is *displayed* on the screen by the user interface as a viewable web page. Normally, the layout and rendering engine takes a snapshot of the DOM information it needs and performs the rest of the work asynchronously; however, it can also be invoked synchronously when JavaScript makes use of APIs that query layout information.

JavaScript Engine. The ZOOMM *JavaScript engine* executes all JavaScript code. The engine's novel design is outside the scope of this paper. Instead, we focus on the integration with the rest of the browser architecture (Section 5).

User Interface. The ZOOMM browser is currently available on Android, Linux, and Mac OS X platforms, and is mainly implemented in platform agnostic C++. For concurrency, we use a custom asynchronous task library, that provides similar functionality to Intel Thread Building Blocks [16]. On Android, a thin Java wrapper is used to create the user interface. User interactions such as touching a link on the display are translated into JNI method calls, which ultimately create work items in the DOM dispatcher. Drawing to the display is performed using the Android NDK, which provides direct access to Android bitmaps. On Linux and Mac OS X, a similar wrapper is implemented in C++ using the Qt interface toolkit [15]. Although our deployment targets are Android devices, the Qt implementation allows much easier debugging and testing on desktop based machines, and the ability to evaluate concurrency beyond what Android devices offer today.

In the following sections, we discuss some of ZOOMM's components in more detail.

3. Aggressive Resource Prefetching

Mobile devices commonly experience high latency when requesting the resources that form an HTML document. In order to reduce the overall time taken to load a page, fetching all of the dependencies from the network as early as possible is very important. This section describes the techniques we employ to prefetch resources discovered in HTML and CSS content.

3.1 HTML Prescanning

Due to idiosyncrasies in the HTML5 specification, the HTML5 parser must wait for `<script>` blocks to finish executing before it can continue parsing. Thus, if a web page references an external resource *after* a script element, fetching the resource cannot be overlapped with the waiting. Potentially, this can delay the completion of page loading. The Mozilla Firefox browser [8] mitigates such situations by speculatively parsing ahead of script blocks to discover new resources. (It may then be forced to throw away some of that work if, for example, JavaScript inserts new content into the DOM tree via the `document.write()` API.) Once resources are discovered, network latency can be masked by requesting multiple resources to be fetched in parallel. This strategy also helps to utilize all available bandwidth. In either case, it reduces the overall time spent waiting for resources to arrive.

In ZOOMM, we favor concurrency to achieve the same goal by running an HTML *prescanning* component in parallel to a (non-speculative) HTML parser. The main objective of the HTML prescanner is to quickly determine all external resources in an HTML document and trigger their fetching from the network. The most commonly referenced resources are images, CSS stylesheets, and JavaScript sources. In addition, stylesheets and JavaScript sources can themselves reference further external resources. Furthermore, the prescanner obtains all `id`, `class` and `style` attributes used in the document.

As network packets of an HTML document arrive, they are given to the prescanner and the actual HTML parser independently. The prescanner is able to run ahead of the HTML parser because it only has to approximately parse HTML in order to find resources, thus skipping the complex DOM tree construction phase. More importantly, the prescanner does not have to wait for the execution of `<script>` blocks to finish.

The processing of prefetched resources works as follows. Images are fetched concurrently with the rest of the page processing. Once downloaded, image data is given to a thread pool for decoding, concurrently. The decoded image is added to the *DOM dispatcher queue*, which updates the corresponding `img` tree node. Then, the image is removed from the set of pending images.

3.2 CSS Prefetching

CSS stylesheets are dispatched to a thread pool responsible for parsing CSS concurrently. If a CSS rule contains further external resources, the parser makes a decision whether to initiate prefetching for them, based on the likelihood that they are actually referenced in the HTML document.

It is crucial to download just enough of the referenced resources. Downloading too little means that new resources are discovered only when styling the DOM tree later on, which incurs additional latency penalties. It is common practice among websites to reference many more resources than are actually needed for any given document, for example by using a site-wide common style file. Downloading all resources invariably consumes too much bandwidth and slows down page loading.

In ZOOMM, the CSS parser employs the `id` and `class` attributes discovered by the HTML prescanner to determine if a rule is likely to be matched. If all of the attribute values referenced in a CSS rule selector have been seen by the HTML prescanner, we assume that the rule will match at least one DOM tree element, and initiate downloading its resources. This heuristic is simple, but effective (Table 1). Note that wrong decisions here do not affect correctness; any missed resources will be discovered during the styling phase, at the cost of additional latency.

3.3 Limitations

ZOOMM's prescanner is limited to discovering resources which can be determined without having to execute JavaScript. Furthermore, the CSS parser may erroneously initiate prefetching of a resource (*false positive*). For example, this occurs if all class IDs were detected by the HTML prescanner individually, but in the HTML document they do not appear nested in the same way as described by any CSS rule. The CSS prefetching algorithm does not generate *false negatives*, except when JavaScript dynamically changes the DOM tree.

4. The CSS engine

Cascading Style Sheets (CSS) is a language used to describe the look and formatting of web sites, thus separating the presentation of a document from its content. Each style sheet consists of an ordered collection of rules with the following format:

```
selector {
        property_1: value;
        ...
        property_n: value;
}
```

For example, the following CSS code makes the browser render all `<cite>` elements whose direct ancestor is a `<p>` element using a white foreground:

```
p>cite { color: white; }
```

It is common for web sites to use several thousand such rules.

ZOOMM's CSS engine performs three jobs: CSS resource prefetching, CSS parsing, and DOM styling. We describe CSS prefetching in Section 3.2. We now describe the other two.

4.1 Concurrent CSS Parsing

During CSS parsing, the CSS engine reads the CSS code and creates a collection of data structures that we call *in-memory rules*. CSS code can be embedded in HTML or linked as separate files, perhaps stored on different servers. Traditional CSS engines — like the ones in WebKit or Firefox— parse CSS sequentially in the main browser thread. Thus, if a page uses embedded CSS, the HTML parser cannot parse the rest of the HTML document until the CSS engine has parsed the style element in the document's header. Moreover, if a page uses several CSS files, they will all be parsed sequentially, even though idle CPU cores are available. Such serialization is particularly noticeable in sites using large CSS files; for example, the main CSS file for BBC News[1] is 250 KBytes in size.

ZOOMM's CSS parser is re-entrant so that it is possible to invoke it from asynchronous, concurrent tasks. During page load, ZOOMM's HTML parser spawns a CSS parsing task for each style element in the DOM tree. Similarly, the resource manager spawns a CSS parsing task for each CSS file it receives. Effectively, this means that a CSS parser instance executes as soon as any new CSS is available, regardless of whether the HTML parser or other instances are already executing. However, ZOOMM must ensure that the total order of the in-memory rules is equal to the one that would have been generated by a sequential CSS engine. Each parsing task receives a unique, sequential ID that is later used to recreate the ordering of the style sheets in the original document.

4.2 Parallel DOM Styling

DOM styling is the means by which the CSS engine uses in-memory rules to determine the style of the nodes in the DOM tree. For each node, the CSS engine must first find all the rules whose selectors match the node, or *rule matching*. Rule matching often returns many — and usually conflicting — rules per node. Using *cascading*, the CSS engine assigns weights to rules and chooses only the ones with the greatest weight. During *style creation*, the CSS engine creates the style data structure using the rules selected by the cascading algorithm and attaches it to the node.

A key insight is that it is possible to concurrently style several DOM nodes as long as certain dependencies are enforced, and we developed a new parallel DOM styling algorithm that leverages it.

Algorithm 1 shows that the CSS engine uses two types of tasks per node to style the DOM tree: *matching tasks* and *styling tasks*.

[1] http://www.bbc.co.uk/news/

Algorithm 1 Parallel DOM Styling Algorithm

```
 1: function StyleNode($node, ruleset$)
 2:     $finalRuleset \leftarrow Cascade(ruleset, node)$
 3:     $node.style \leftarrow BuildStyle(finalRuleset, node)$
 4: end function
 5:
 6: function MatchNode($node, ruleset, styling$)
 7:     $child \leftarrow node.FirstChild()$
 8:     while $child \neq NULL$ do
 9:         $r \leftarrow NewRuleset()$
10:         $ts \leftarrow NewTask(StyleNode(child, r))$
11:         $tm \leftarrow NewTask(MatchNode(child, r, ts))$
12:         $tm.SetSuccessor(ts)$
13:         $styling.SetSuccessor(ts)$
14:         $tm.Spawn()$
15:         $ts.Spawn()$
16:         $child \leftarrow child.NextSibling()$
17:     end while
18:     $Match(node, ruleset)$
19: end function
20:
21: function StyleDomTree($tree$)
22:     $root \leftarrow tree.root()$
23:     $ruleset \leftarrow NewRuleset()$
24:     $ts \leftarrow NewTask(StyleNode(root, ruleset))$
25:     $tm \leftarrow NewTask(MatchNode(root, ruleset, ts))$
26:     $tm.SetSuccessor(ts)$
27:     $tm.Spawn()$
28:     $ts.Spawn()$
29:     $WaitForTasks()$
30: end function
```

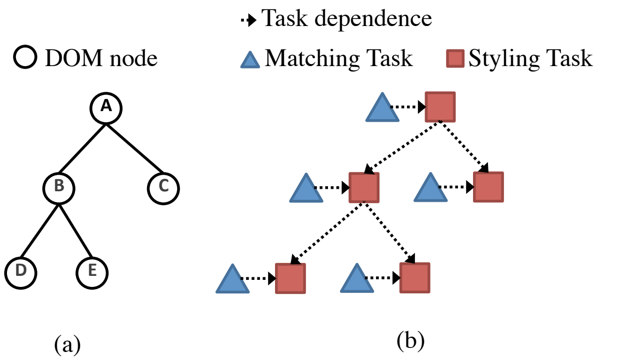

Figure 4. DOM tree (a) and corresponding task DAG (b).

Matching tasks start by spawning new matching and styling tasks for each of the node's children. Then, they rule-match the node. Their output is a set of rules that are applicable to the node. They spawn children tasks before they do the actual work because styling tasks are fully independent, and it is desirable to have as many of them executing as possible.

Styling tasks apply the cascading algorithm and create the final style data structure for each node. A styling task must satisfy two dependencies before it can execute. First, it can only execute after the matching task working on the same node has completed execution, since the cascading algorithm uses the rules selected by the matching task. Second, a styling task working on a node can only execute after the styling task working on the node's parent has completed execution. This is because some style properties of a node may be inherited from its parent. By using two types of tasks, our algorithm can rule-match a node before its parent is styled. Figures 4a and 4b show a DOM tree and its corresponding task DAG, respectively.

This basic version of the algorithm limits style sharing to parent-child sharing. However, we subdivide style objects into substyles containing related properties, and allow sharing at the substyle level, which increases the degree of available sharing (see Section 6.1). For example, if a child node uses the same font properties as its parent, then they can share the font substyle. To add support for sibling style sharing, the matching task must speculate whether a child node may be able to share its style with a previous child node before spawning tasks for it. If the answer is yes, it does not spawn new tasks for the second child.

Matching tasks can be expensive because the rule matching algorithm must decide whether each selector applies to the node or to any of its ancestors. This may require traversing the tree all the way to the root. For example, rule-matching BBC News requires more than 400,000 of such walks. WebKit saves 90% of them by using a Bloom filter [3] that stores information about the ancestors of a DOM node. In Safari 5.0, a Bloom filter instance amounts to a space overhead of 4 KB. ZOOMM cannot use the same data structure because it would require a new Bloom filter instance per matching task. Instead, ZOOMM utilizes *matching bitmaps*. Matching bitmaps are fixed size (64 byte) and record whether an ID, a class or a tag has been seen in one of the node ancestors.

Section 3.2 describes how ZOOMM uses element `id`, `tag` and `class` attributes to predict whether an image referenced in the CSS file should be prefetched. These attributes are stored in a database that sorts them according to the number of times each one appears in the document. Before the rule matching algorithm starts, the CSS engine assigns a bit to each of them in a bitmap data structure. If the number of ids and classes is larger than the bitmap size, a single bit can be assigned to multiple items. During rule matching, each matching task receives a matching bitmap from its parent. Matching tasks use the matching bitmap to filter out rules that could never match: if the bit corresponding to a tag, id or class is not set, that means that no ancestor has it. Therefore, there is no need to traverse the tree. After matching is completed, matching tasks add their node's id, class, and tag to the bitmap and send a copy to their descendants.

5. JavaScript Engine

The ZOOMM browser includes a JavaScript engine that is optimized for parallel execution. In particular, our engine expoits concurrency by compiling multiple scripts in parallel, as well as compiling scripts asynchronously with the rest of the browser passes. To achieve this, the JavaScript engine uses a thread pool and the JIT compiler uses separate state for each script.

Due to JavaScript semantics, execution of scripts is performed sequentially in the main engine thread. When the HTML parser or the DOM dispatcher (e.g., for user interface events) requests the execution of a JavaScript script that has not been compiled already, compilation is initiated. In either case, the engine waits for the compiled result, and then executes the script. The goal of the engine is to use available resources on the platform to improve the generated code for JavaScript execution.

For adaptive compilation and execution of the JavaScript code, the JavaScript engine consists of an interpreter, and two compilers:

Interpreter. The interpreter is used for fast startup execution of small JavaScript scripts. It is mainly invoked on page load, since many pages have inline JavaScript code that is executed only once and mainly calls into the browser bindings.

Light compiler. This compiler is optimized for page load and generates executable code for infrequently reused JavaScript code. We have a parametrized threshold that triggers the invocation of the light compiler.

Full compiler. This compiler is optimized for interactivity and *web apps*, and generates higher quality code for heavily reused JavaScript code. The slower code generation of the full compiler is amortized between multiple runs of the reused code. Compared to the light compiler, this compiler achieves significant speedup for iterative web apps. For example, using this compiler, an *N-body simulation* web app runs six times faster than the same application compiled with the light compiler.

A full description and evaluation of the JavaScript engine is outside the scope of this paper.

6. Layout, Rendering, and Display

Once styles have been applied to the DOM tree by the CSS engine, it is necessary to use this information to produce a bitmap image that will be displayed to the user. This job is performed in four steps:

1. The *layout tree* is created or updated by the page event loop. This tree captures all the information from the DOM that is relevant for the subsequent steps; each node represents a visual element on the web page. Only this step needs to run on the page event loop; after the layout tree is created, the remaining steps are independent of the DOM, and can run concurrently with tasks like JavaScript.
2. The *layout engine* is responsible for solving the system of constraints expressed by the layout tree's styles and structure; it implements the CSS layout algorithm [6]. Ultimately it annotates each layout tree node with width, height, margins, and other spatial information which determines how it will be displayed on the screen.
3. The *rendering engine* walks the annotated layout tree nodes and draws them, along with any text or other content they may contain, into a bitmap. The resulting bitmap is again independent of the previous steps, and further manipulation of it can happen concurrently.
4. The *user interface*, running on its own thread to assure responsiveness to user input, accepts the bitmap produced by the rendering engine and displays it on the screen. The user interface is also responsible for scrolling; once a bitmap has been displayed, the user can scroll freely even if all of the other threads in the system are busy.

Layout, rendering, and display thus involve three different tasks in a pipeline-like structure, although as we discuss below, there are subtleties that may increase its complexity. This arrangement isolates both the page event loop and the user interface from the delays caused by layout and rendering, allowing JavaScript and interactive events like scrolling to run as fast as possible.

In the following subsections we explore the detailed design of each step in this process.

6.1 Layout Tree

The layout tree's purpose is to decouple layout, rendering, and display from the page event loop, allowing JavaScript and other types of tasks to run freely. The main obstacle to this goal is JavaScript's ability to change DOM nodes and CSS rules at will; these changes may cause inconsistent layout results or rendering artifacts if they are not protected against.

The CSS layout rules do not create a one-to-one mapping between DOM nodes and layout nodes; this means that a distinct tree must be created for the layout nodes[2] regardless of our concurrency goals. To eliminate any possible interference from JavaScript, we augment the information normally found in the layout tree in such a way that there is no longer any need to refer back to the DOM or the CSS rules.

We use different strategies for different types of information, as follows:

CSS Styles. Because of their large size, we divide style objects into a number of substyles, each containing properties that usually vary together. Each substyle can be shared with many style objects using copy-on-write semantics; a style object thus consists of a set of references to potentially-shared substyles. In addition to greatly reducing memory usage, this approach has the advantage that we do not need to worry about modification to styles when building the layout tree; as long as our style object has a reference to a substyle, we know that it will remain unchanged, regardless of any concurrent style changes that may occur.

DOM Text Content. It would be feasible to use a copy-on-write approach for text as well, but because text content was so small compared to other types of data on our test sites, we simply copy the text into the layout tree.

DOM Image Content. Images are immutable in ZOOMM, so all we need to place in the layout tree is a reference.

Canvas Elements. Canvas elements need to be treated specially because their contents are often changed rapidly by JavaScript. Given that they contain image data that is fairly expensive to copy and that the old contents of a canvas element are only occasionally needed, a copy-on-write approach is not appropriate here. Instead, it proves cheaper to copy the contents of the canvas into a corresponding buffer in the layout tree. These buffers are reused to minimize memory allocation, and we use dirtiness annotations in the DOM to avoid copying if the canvas element contents have not changed.

An alternate strategy would be to avoid copying the canvas contents at all and simply paint whatever contents the canvas has at render time; this has the disadvantage that tearing will be visible to the user. Effectively, copying the canvas contents amounts to double buffering.

HTML Attributes. Only a small subset of HTML attributes matter to layout; these are presentational values like `border`. In most cases these map directly to CSS properties, and when these attributes are set on a node we translate them into changes to that node's style. Certain presentational HTML attributes have no CSS equivalents; for these, we make use of special ZOOMM-only CSS properties.

One interesting case concerns the HTML `height` and `width` attributes; although in many cases these are equivalent to the corresponding CSS properties, for canvas elements the HTML attributes control the dimensions of the canvas's internal coordinate system, while the CSS properties control the scaling of the canvas on the page. This difference requires that we record both values in the style for canvas elements.

Conceptually, a new layout tree is generated whenever the DOM or CSS are updated. This is usually triggered by JavaScript, but may also be the result of the arrival of new resources. However, in practice we do not need to generate a new layout tree unless we are going to use it, so we defer the work until either we reach the point of displaying the previous tree and detect that the page's contents have changed, or we must compute layout to determine a value that JavaScript is requesting.

[2] Although in some implementations, a single node can serve both purposes when a one-to-one mapping is possible.

We have found that in practice, JavaScript often makes numerous references to values computed by layout during page load. Current browsers typically only allow one layout tree to exist at a time, synchronizing it with the DOM according to some policy. This is not a problem if layout blocks the page event loop anyway, but in ZOOMM we run layout asynchonously; if we were forced to wait for layout and rendering to finish to update the layout tree, we might block the event loop for the time required to run layout twice – one time to be able to update the layout tree, and a second time to actually run the layout algorithm. We avoid this problem by allowing multiple layout trees to exist at the same time, and running layout in the page event loop thread if up-to-date layout information is not available when JavaScript requests it.

To minimize the costs of copying or updating our layout trees – indeed, these are the same operation for us – we treat the layout trees as an immutable data structure. Information that is likely to change frequently is factored out of the layout tree nodes and stored in separate data structures at the root; the nodes themselves only store keys or indices into these data structures. This arrangement means that we often only need to update a small portion of the overall data in the layout tree when something changes; though we only handle certain common cases at present, this approach can be taken quite far. As a concrete example, we store the contents of canvas elements in a list, while canvas nodes in the layout tree contain only an index into this list. When the canvas contents are updated, the new layout tree can share all of the layout nodes with the old tree; only the list of canvas contents must be duplicated.[3] Note that there are some data structures in the layout tree that are not truly immutable until the layout algorithm has already been run; we take note of whether this has happened and decide whether to copy those data structures on that basis.

Creating or copying the layout tree is the only step that must take place in the page event loop. This prevents JavaScript and CSS from running concurrently and ensures that the resulting layout tree is consistent. Once the layout tree is ready, we transfer its ownership to the task responsible for layout and rendering and the event loop can continue processing without any further delay.

6.2 Layout Engine

When the layout engine, running concurrently with the page event loop, receives a layout tree, it consults metadata stored in the layout tree to decide whether to perform CSS layout [6]. This information is stored as part of the process of creating or updating the layout tree, since inferring it after the fact is much more difficult.

Currently we use coarse-grain *dirty bits* to record which aspects of the page have changed, allowing us to determine whether a layout is necessary. In some cases, such as when only images or the contents of canvas elements have changed, the information in the layout tree will be up to date and there will be no need to perform layout at all. In all other cases we currently run a full layout. It is possible to determine the scope of the effect of particular DOM changes and only run the layout algorithm on the subtrees affected; we leave this for future work.

The particular algorithm used to compute layout is orthogonal to our asynchronous layout approach. We use asynchrony to move layout and rendering out of the critical path of the page event loop; a parallelized layout algorithm complements our current design by reducing the delay until layout changes are visible on the screen. Additionally, JavaScript queries like those we mention above sometime force layout to be performed in the page event loop; this case

[3] Indeed, using standard immutable data structure techniques, only updating part of the list is necessary. Our current implementation uses a coarser granularity, however.

would also benefit from a parallelized layout algorithm. We leave this extension for future work.

Regardless of the layout implementation, after the layout engine is finished each node in the layout tree is annotated with spatial information such as its final x and y position and layer. This annotated layout tree is then transferred to the rendering engine.

6.3 Rendering Engine

The rendering engine also runs concurrently with the page event loop, after the layout task has executed. It walks the annotated layout tree and paints the contents of the page into a bitmap. We use a simple sequential algorithm that performs rendering according to the CSS standard [6]. It would be possible to use many parallel walks over the layout tree to render into independent tiles simultaneously, but we leave an investigation of this approach for future work; because it does not block the page event loop, rendering is currently not a bottleneck in ZOOMM.

Beyond parallelism, an additional advantage of the layout tree design in ZOOMM is that we can treat layout and rendering as a service which is shared between web pages. Since layout trees do not refer back to the DOM or CSS they were constructed from, it's no problem for the same layout and rendering thread to handle all layout trees regardless of their source. This means that expensive, finite rendering-related resources like bitmaps only need one instance per browser window.

We reuse the same bitmap as long as change to the environment like a resize of the browser window does not invalidate it. This greatly reduces ZOOMM's memory requirements, but it means that we must ensure that the bitmap is copied into graphics memory before any further rendering is performed. This is accomplished by temporarily transferring control of the bitmap to the user interface. When the user interface finishes processing the bitmap, rendering is complete and the layout and rendering thread notifies the page event loop that it is ready to accept any updated layout trees that become available.

6.4 User Interface

The user interface runs on its own thread. The layout and rendering thread transfers control of the bitmap representing the current page contents to the user interface, and then blocks while the user interface uploads the bitmap into graphics memory. This task is performed in the user interface thread only because of platform-related restrictions on which thread must perform this action. After this is complete, the user interface thread notifies the layout and rendering thread that it may continue working, and displays the new content to the user.

The user interface thread is also responsible for processing events caused by user interaction. Most events are handled by JavaScript; these are packaged and delivered to the appropriate page event loop. However, certain events can be handled by the user interface directly. For example, because a page's contents are stored in graphics memory, the user interface can scroll the contents directly. This means that even if the page event loop or layout and rendering thread are busy, scrolling continues to be smooth and responsive.

7. Related Work

There is a wide variety of existing studies on browser parallelism and concurrency in the literature. We summarize the most relevant of these here.

Process-Per-Page Browsers. Chrome [17], the WebKit2 [24] engine, and Gazelle [21] exploit parallelism by using separate processes for the event loop of each open page, essentially delegating the responsibility of using multiple cores to the OS. This is simple to implement and provides isolation between different sites. However, processes are heavyweight in terms of both memory and startup overhead, and page-level parallelism doesn't address the needs of mobile browsers, where single-page performance is often inadequate and users do not open many tabs at once.

Chrome additionally performs display in a separate process [5]; layout and rendering take place in the page event loop, and the resulting rendered segments of the page are transferred to a separate process, allowing the segments to be displayed and scrolled without blocking the event loop. In contrast, our model moves layout and rendering almost entirely out of the page event loop, allowing more time for handling UI events and executing JavaScript.

OP and OP2. Grier et al. [10] developed browsers designed for security and isolation. The OP and OP2 browsers use a per-page multiprocess architecture that places several browser components, such as networking, in different processes. They observed speedups from the ability to overlap operations at the process level. While a multicore/parallelism study was not their primary goal, the observations made are a good indication of the potential of overlapping browser components. We improve on their multiprocess model by adding algorithm-level parallelism and separating out more components, such as layout and rendering, which can run asynchronously in parallel with other activity on the page.

Adrenaline. Mai et al. [12] speed up page processing by splitting the original page in minipages. Each minipage contains only a subset of the CSS and HTML of the original page. This makes CSS and layout faster because the DOM trees have fewer nodes and the style sheets have fewer rules. Additionally, this work can happen in parallel with JavaScript, which runs in a process associated with the main page; Adrenaline is the only other browser we are aware of that allows this. However, Adrenaline must merge other mini pages into the master page if JavaScript makes any reference to their content, preventing any further parallelism for that mini page. In contrast, JavaScript's DOM access in our browser is unrestricted and does not penalize parallelism or trigger the overhead of page merging. Our method also does not require server-side support as Adrenaline does.

The Berkeley Parallel Browser Project. Meyerovich and Bodik have looked at the problem of parallelizing the CSS and layout algorithms [13]. Their initial results are very promising, offering $80\times$ speed-up in some instances. However, their CSS algorithm only supports a subset of the CSS standard, and it is a standalone component that only performs selector matching. ZOOMM's parallel styling algorithm is CSS 2.1 compliant and combines parallel matching with parallel cascading. Their study also addresses the problem of parallel layout. Their work is orthogonal to ours – we address performing layout entirely outside of the page event loop and could use their parallel algorithm.

Although they target only a well-behaved subset of the specification, their experience shows that browser processing algorithms can be made highly concurrent if one could modify the standard. We expect that many of their optimizations would be beneficial if combined with our approach, although we leave this for future work.

Parallel CSS in Firefox. Badea et al. [2] profiled Firefox and found that, in most cases, rule matching a node requires rule matching all the node's ancestors. They propose using helper threads to speed up matching of ancestor selectors. In their implementation, each helper thread is assigned a number of ancestors to process. WebKit and ZOOMM avoid most of those traversals by using bloom filters and matching bitmaps, respectively, addressing this issue in a sequential fashion.

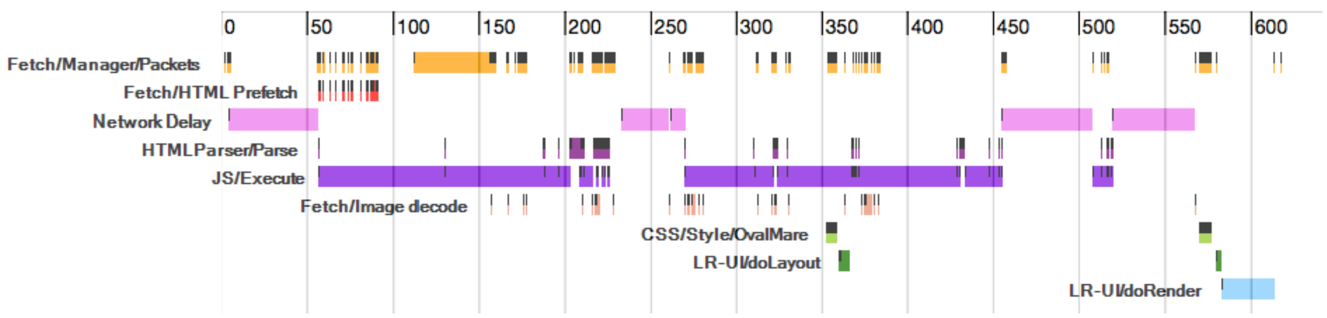

Figure 5. Gantt chart of loading the Yahoo! web page with ZOOMM from a mirror: concurrency among components are visible through overlapping bars; intra-component concurrency is not shown here.

Prefetching. There is a long history in web browsers of resource prefetching. A variety of schemes has been developed over the years, with and without server support, with and without prefetch directives, augmented with features like learning components, etc.. We refer to Duchamps and Fisher et al. for summaries [7, 9].

The HTML parser in Mozilla Firefox 4 and later supports *speculative parsing* [14]: instead of waiting for the execution of JavaScript to finish, the parser scans ahead for external resources (scripts, style sheets and images) and downloads them. The approach is different than in ZOOMM, in that Firefox also continues to construct the DOM tree. This work is necessarily speculative in nature, since the currently executing JavaScript can alter the parsing context (e.g., by emitting new content via `document.write()`). In that case, the speculative work has to be undone. In ZOOMM, we chose to make the HTML prescanner independent from the parser, thus simplifying the architecture of both. Furthermore, it allows the prefetcher to be moved into a separate thread, thereby hiding any overhead incurred as long as enough compute resources are available. However, our approach precludes speculative construction of the DOM tree. As future work, it would be interesting to compare the benefits of the Firefox approach in more detail to ZOOMM's approach, and investigate whether it can be incorporated into ZOOMM's architecture.

Compared to Firefox and earlier approaches, the novelty in our approach lies not in the prefetching itself, but in the additional funneling of information from the HTML prefetcher to the CSS parser, thereby allowing resources referenced in CSS documents to be requested ahead of time.

8. Experimental Results

In this section we present data to demonstrate the benefits of concurrency with respect to hiding the network latency, parallel CSS styling, and image decoding. We show two sets of data, one set to demonstrate effectiveness on a mobile platform (an HTC Jetstream device using a dual-core Snapdragon processor) and another set (on an Intel 6-core Xeon) that demonstrates the scalability of our design targeted at future mobile devices.

Performance. In order to test the overall performance of the ZOOMM browser, we used the publicly available Vellamo [20] benchmark suite. Vellamo tests the overall page load time, from the initial fetch to showing pixels on the display for a set of web sites: CNN, BBC, Yahoo!, Guardian, New York Times, Facebook, Engadget, and QQ (also shown in Table 1). Vellamo uses the user agent of a desktop browser to ensure that the HTML is not simplified for a mobile device. Pages are loaded and displayed twice: first time with a cold cache, and second using the browser cache. We measured Zoomm against the standard WebKit-based browser included with Android on the HTC Jetstream device. The Vellamo test was performed using a local network proxy to ensure that the network conditions were consistent between both browsers. Overall, the ZOOMM browser completed the test run in 55 seconds, while WebKit completed it in 113 seconds. Figure 6 shows the time for loading each page in the two browsers. The load times are the average over 5 runs of the non-cached round. While there is a large variability between sites, ZOOMM is always faster, with an overall improvement of approximately 2×, demonstrating the effectiveness of the techniques uses in ZOOMM to improve browser performance.

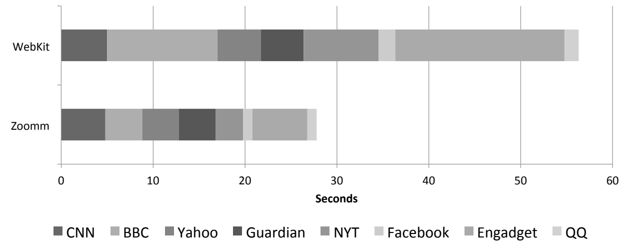

Figure 6. WebKit vs. Zoomm page load time on HTC Jetstream

Concurrency. Figure 5, illustrates the concurrency among the major components in ZOOMM while loading the Yahoo! web page from a mirror over a high-bandwidth/high-latency network connection representative of future 4G networks (Gigabit ethernet with artificial 50ms network latency). In addition, there also intra-component concurrency which is not shown; for example, image decoding and CSS styling internally run with multiple threads.

The "Network Delay" row indicates times when the DOM dispatcher loop sleeps because there are no events (network activity, user interaction, timers, etc.) to process due to waiting for external resources. The first network delay is between the request of the HTML document and the first packets arriving. Then the HTML parser processes the packets and kicks off execution of JavaScript code. The HTML parser waits with further processing until the JavaScript finishes. However, in parallel, further packets arrive and images are being decoded. We note that even for a single run, styling and rendering happens twice, the first time triggered by a call to the `getComputedStyle` function from JavaScript.

Compared to the 2010 WebKit numbers in Figure 2, JavaScript has become more prevalent. Also, ZOOMM's JavaScript engine is comparatively less mature than WebKit's and hence shows up more prominently in the chart.

Prefetching. Figure 7 shows the speed-up gains from early resource discovery on two different networks: the 4G Verizon net-

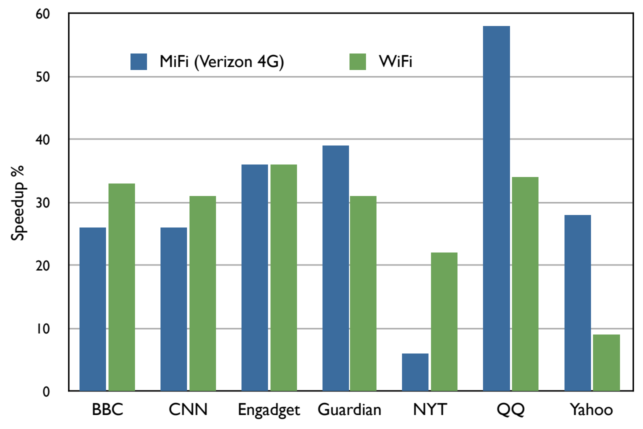

Figure 7. Prefetching speed-up on 4G and WiFi networks (higher numbers are better). The bars show the percentage reduction in page load time when prefetching is enabled in the ZOOMM browser vs. a baseline page load without prefetching enabled.

work through a MiFi device, and a WiFi wireless network. In both cases, devices were connected wirelessly to these access points. The pages are fetched from the real servers on the Internet. Experiments were run with a cleared cache. The speed-up is measured on total page load, with and without prefetching enabled. While there is significant variability in the results due to Internet latencies, we improve page load time in all cases, up to 58% (25% geometrical mean).

Table 1 shows the number of resources that are successfully requested by the prefetching stage, and the number of resources are missed due to use of JavaScript. Note that resources would also count as "missed" if the prefetching algorithms would fall behind the actual HTML and CSS parsers. However, we found this to be never the case in all our experiments. The prefetching components are fast enough to always finish much earlier than the parsers.

Despite the heuristic nature of some of the prefetching decisions, they are quite accurate: in our experiments, 80–95% of all externally referenced resources in a document are prefetched correctly, with only a small error rate. Due to bandwidth and power considerations, our heuristics are still conservative, i.e., they rather prefetch too little than too much: "Missed Prefetch" (not prefetched, but needed for rendering the web page) numbers are higher than "Mistaken Prefetch" (prefetched, but not needed for rendering).

CSS Performance. In Figure 8 we present scalability results for the parallel CSS algorithm. The algorithm scales well for complex sites, with large numbers of rules and many DOM nodes. Because the testing machine has 6 hyper-threaded cores, we observe limited scalability beyond 6 cores. Our measurements indicate that ZOOMM spends 97% of CSS styling time executing matching tasks, and only 3% executing node styling tasks.

Table 2 shows the accuracy of ZOOMM's matching bitmaps technique for a set of websites. On average, ZOOMM avoids 90% of the walks to the root of the DOM tree, with only 0.024% of false positives. False positives occur because matching bitmaps do not record the order in which labels and ids are encountered. For example, suppose ZOOMM wants to know whether the following rule applies to a `<p>` node: `h1>h2 p {color: red}`.

Assume that the matching bitmap indicates that both `<h1>` and `<h2>` are `<p>`'s ancestors. ZOOMM's matching algorithm must traverse up the tree to check whether there is a `<h1>` node that is a direct ancestor of a `<h2>`'s. If that is not the case, then it is a false positive. Note that false positives do not cause the page to render incorrectly, they just waste CPU cycles.

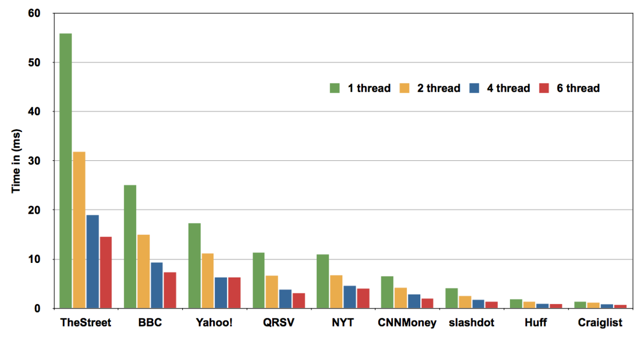

Figure 8. Large and complex web sites benefit from the scalability of the parallel DOM styling algorithm, here running on an Intel Xeon with 6 hyper-threaded cores. The first five web sites were retrieved using a desktop user agent, making the HTML more complex. The last five web sites were retrieved using a mobile user agent, where the server sends simpler HTML for display on a small mobile device. Desktop-based HTML experiences larger improvements in performance due to the size of the content.

Site	% Avoided Walks	% False Positives
CNN	88.94	0
BBC	88.02	0.021
Yahoo!	86.83	0.080
Guardian	94.19	0
NYT	98.74	0.035
Engadget	82.77	0
QQ	92.46	0.031
Average	90.28	0.024

Table 2. Accuracy of the matching bitmaps technique for the Vellamo benchmark with desktop user agent. Higher number of avoided walks is better, and lower number of false positives is better.

Parallel Image Decoding. Figure 9 shows the scalability of offline image decoding when increasing the number of threads in the shared thread pool used for CSS prefetching and image decoding. In the figure we average the total time over 5 runs on a test page that consists of 36 images of 1.6 Megapixels each. These plots demonstrate that with enough work, we can take advantage of multiple cores for image decoding. However, other factors, like memory and network bandwidth, as well as server latency, and number of allowable connections per server, can lead to diminishing returns. In our experiment, we did not observe speed-ups beyond six cores, due to the hyper-threaded nature of our test CPU, and the browser computations unrelated to image decoding.

9. Conclusions

In this paper we presented ZOOMM, a parallel browser engine designed to exploit multicore concurrency. We demonstrated how concurrency helps hide up to 60% of the network latency when loading web pages. We also demonstrated that scalable concurrency exists and can be efficiently exploited for a number of browser algorithms — CSS styling and image decoding are only two such examples. When loading pages, we demonstrate that ZOOMM is twice as fast as a native WebKit browser on a dual-core Android platform. The technologies demonstrated in ZOOMM are being adapted for production browsers.

Web site	Correct Prefetch HTML		Correct Prefetch CSS		Missed Prefetch		Mistaken Prefetch HTML		Mistaken Prefetch CSS		Total Resources	
	Files	Bytes	Files	Bytes	Files	Bytes	Files	Bytes	Files	Bytes	Files	Bytes
cnn.com	34	979,695	52	409,377	2	372	0	0	5	3,371	93	1,392,815
bbc.co.uk/news	54	610,479	24	407,819	16	468,371	0	0	1	1,277	95	1,487,946
yahoo.com	44	672,595	13	264,603	2	2,016	1	0	0	0	60	939,214
guardian.co.uk	49	1,018,738	14	92,997	7	102,087	1	0	3	11,305	74	1,225,127
nytimes.com	73	1,046,636	9	73,487	13	228,162	1	10,837	1	89	97	1,359,211
engadget.com	128	2,023,135	84	651,030	5	104,320	0	0	9	34,824	226	2,813,309
qq.com	45	485,264	22	167,078	7	39,361	0	0	0	0	74	691,703

Table 1. Combined HTML & CSS Prefetching initiates download of most external resources ahead of their discovery by the HTML and CSS parsers with high accuracy ("Correct Prefetch") and small error ("Missed/Mistaken Prefetch"); web sites from *Vellamo* benchmark. "Total Resources" denotes the number of *referenced resources* in a web page.

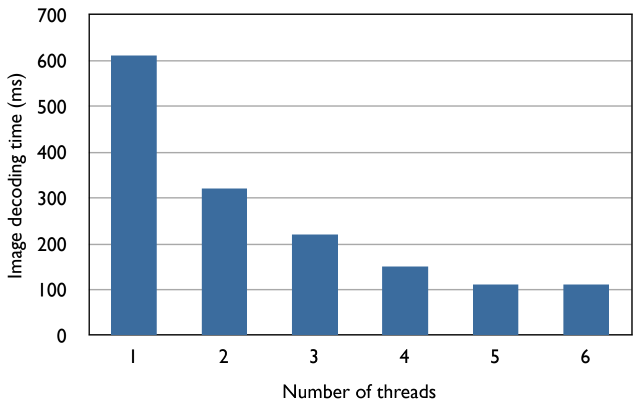

Figure 9. Image-heavy web site benefit from the scalability of the parallel image decoding algorithm, running on an Intel Xeon with 6 hyper-threaded cores.

Acknowledgments

We would like to thank Nayeem Islam and the Qualcomm Research Executive team for the opportunity to build the ZOOMM prototype. We thank Alex Shye, Greg Wright, Dario Gracia, Christoph Kerschbaumer, and Madhukar Kedlaya for their contributions. We also thank the reviewers for their thoughtful comments.

References

[1] D. Astely, E. Dahlman, A. Furuskar, Y. Jading, M. Lindstrom, and S. Parkvall. LTE: the evolution of mobile broadband. *IEEE Communications Magazine*, 47(4):44–51, Apr. 2009.

[2] C. Badea, M. R. Haghighat, A. Nicolau, and A. V. Veidenbaum. Towards parallelizing the layout engine of firefox. In *Proceedings of the Second USENIX Workshop on Hot topics in Parallelism, HotPar*, pages 1–6, 2010.

[3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[4] The Google Chrome web browser. `https://www.google.com/chrome`.

[5] The chromium projects: Compositor thread architecture. `http://dev.chromium.org/developers/design-documents/compositor-thread-architecture`.

[6] Cascading style sheets level 2 revision 1 (css 2.1) specification. `http://www.w3.org/TR/CSS2`.

[7] D. Duchamp. Prefetching hyperlinks. In *USENIX Symposium on Internet Technologies and Systems*, pages 12–23, October 1999.

[8] The Mozilla Firefox web browser. `https://www.mozilla.org/firefox`.

[9] D. Fisher and G. Saksena. Web content caching and distribution. chapter Link prefetching in Mozilla: a server-driven approach, pages 283–291. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[10] C. Grier, S. Tang, and S. T. King. Designing and implementing the OP and OP2 web browsers. *ACM Transactions on the Web (TWEB)*, 5(2), May 2011.

[11] I. Hickson. HTML5 specification, March 2012. `http://whatwg.org/html`.

[12] H. Mai, S. Tang, S. T. King, C. Cascaval, and P. Montesinos. A case for parallelizing web pages. In *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism*, HotPar'12, Berkeley, CA, USA, June 2012. USENIX Association.

[13] L. A. Meyerovich and R. Bodík. Fast and parallel webpage layout. In *Proc. of the Intl. Conf. on the World Wide Web*, pages 711–720, 2010.

[14] Mozilla Developer Network. Optimizing your pages for speculative parsing. `https://developer.mozilla.org/en-US/docs/HTML/Optimizing_Your_Pages_for_Speculative_Parsing`, 2012.

[15] Nokia. Qt - cross-platform application UI framework. `http://qt.nokia.com/`, 2012.

[16] J. Reinders. *Intel Threading Building Blocks: Multi-core parallelism for C++ programming*. O'Reilly, 2007.

[17] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, pages 219–232, New York, NY, USA, March 2009. ACM.

[18] V. Roto. *WEB BROWSING ON MOBILE PHONES - CHARACTERISTICS OF USER EXPERIENCE.* PhD thesis, Helsinki University of Technology, 2006.

[19] The Safari web browser. `http://www.apple.com/safari/`.

[20] The Vellamo mobile web benchmark. `http://www.quicinc.com/vellamo/`.

[21] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal os construction of the gazelle web browser. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 417–432, Berkeley, CA, USA, August 2009. USENIX Association.

[22] Z. Wang, X. Lin, L. Zhong, and M. Chishtie. Why are web browsers slow on smartphones? In *Proc. ACM Intl. Workshop on Mobile Computing Systems and Applications (HotMobile)*, March 2011.

[23] The WebKit open source project. `http://www.webkit.org`.

[24] WebKit2. `http://trac.webkit.org/wiki/WebKit2`.