Towards Parallelizing the Layout Engine of Firefox*

Carmen Badea
UC Irvine
cbadea@uci.edu

Mohammad R. Haghighat
Intel Corporation
mohammad.r.haghighat@intel.com

Alexandru Nicolau
UC Irvine
nicolau@ics.uci.edu

Alexander V. Veidenbaum
UC Irvine
alexv@ics.uci.edu

Abstract

The Mozilla Firefox browser currently accounts for ∼25% of the total web browsers market segment share, establishing itself as the second most popular browser worldwide after Microsoft's Internet Explorer. With the recent adoption of a tracing JavaScript Just-In-Time (JIT) compiler in Firefox 3.5, its performance has improved significantly, especially for web pages that make heavy use of JavaScript. Currently, the heavy performance hitter component of Firefox is its layout engine. According to our extensive performance measurements and analysis on representative applications running on Intel platforms, the layout engine of Firefox accounts for ∼40% of its total execution time, with the Cascading Style Sheets (CSS) rule-matching process being the hot part of layout. We have developed a tracing and profiling mechanism that has enabled us gain in-depth insight into the CSS rule-checking runtime characteristics. This data has proven extremely helpful in determining that the CSS rule-matching component is a suitable target for parallelization. The collected trace information has also guided us to an effective parallel implementation of the Firefox CSS rule-matching component that delivers large performance speedups on popular web pages, including almost 90% of Mozilla's page-load tests that comprise ∼400 distinct web pages from all over the world. For these tests, the *entire* page-load process shows performance changes ranging from -0.5x to +1.84x, with an average improvement of ∼1.1x, when two worker threads cooperate in doing the layout. To our knowledge, this is the first fully functional parallel implementation of CSS rule matching that delivers significant speedups in a world-class browser.

1 Motivation

Multi-core processors have gained widespread use in general-purpose systems and are expected to proliferate in embedded devices as well. However, many of the popular applications run by end users do not take advantage of the available hardware parallelism, and as such it is common to have computing resources sitting unused.

More recently, with the advancements in the web application technology, the browser has emerged as the favorite platform for deploying sophisticated applications that were traditionally possible only as native desktop applications (e.g., Google Docs and Zimbra Collaboration Suite). However, among the most popular browsers, there are only a few that actually take advantage of the available cores in their various components. The Chrome browser is one such browser, by treating each tab as a separate process (including any JavaScript code being executed in a tab), thus naturally lending itself to parallelism.

Among the open-source browsers, Mozilla Firefox is currently the most widely used one. According to [2], as of April 2010 Firefox accounts for ∼25% of the total browsers market share, second only to Microsoft's Internet Explorer in terms of usage, as shown in Figure 1.

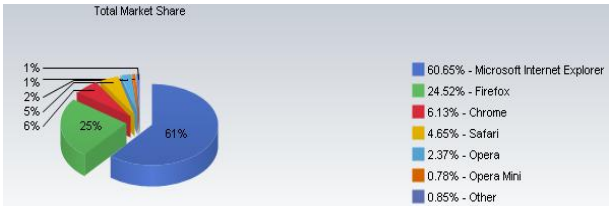

Figure 1: Browser market share distribution, as of April 2010

With the recent adoption of the tracing JIT JavaScript engine (code-named TraceMonkey [5]) in the Firefox 3.5 browser, its performance has improved significantly, especially for web pages that make heavy use of JavaScript. Our extensive performance experiments show that currently the heavy performance hitter component in the Firefox browser in many real usage cases is its layout engine, with the Cascading Style Sheets(CSS) rule-matching component accounting for the largest portion of the layout engine execution time (more details in Section 2.2). Moreover, by incorporating our tracing and profiling mechanism into the Firefox's CSS rule-checking com-

*This work was supported in part by the National Science Foundation under grant NSF CCF-0811882.

ponent, we were able to identify performance bottlenecks and determine that the rule-matching process is a suitable target for parallelization.
The in-depth profiling and tracing data obtained proved to be of utmost importance in our development of the first fully-functional parallel implementation of CSS rule matching in a real browser. As we will show in the upcoming Section 3, our parallelized Firefox prototype yields considerable performance speedups despite thread management overhead and unavoidable speculative execution (for more details, see Section 2.3).
The rest of this paper is organized as follows: in Section 2 we will give a brief overview of how the CSS process works in general and how it is implemented in the Firefox browser in particular. We will also show the profiling and tracing data indicating that the CSS rule-matching component is a heavy performance hitter in Firefox, as well as amenable to parallelization and we will discuss the parallelization strategy we used in developing the CSS rule-matching parallelized prototype in Firefox. Section 3 illustrates the performance speedups obtained by carrying out browsing experiments using our parallelized prototype. Section 4 reviews other projects related to our research, followed by a brief summary of the work presented in this paper in Section 5.

2 Methodology

In this section we will discuss the approach we took for developing a parallelized version of the CSS rule-matching component in Firefox and we begin by giving a brief review of how CSS works in general.

2.1 CSS Overview

Cascading Style Sheets (CSS) [10] represent a stylesheet language used for describing the presentation semantics of web pages. It offers a way of providing clean separation between webpage content (e.g., HTML) and its presentation (e.g., fonts, colors, spacing, etc.). CSS allows multiple webpages to share formatting characteristics, reduces the amount of repetition in specifying layout styles, and makes it easy to apply style changes in the possibly large number of web pages of a given site. The desired layout features are expressed through CSS rules. Figure 2 illustrates the general format of CSS rules, as well as a very simple example: a CSS rule that will apply the black color to the `<body>` element of the web document that uses this CSS declaration, as well a `1em` padding[1]. There are many types of selectors, such as descendant selectors, sibling selectors, child selectors,

[1]Figure courtesy of `http://css.maxdesign.com.au`

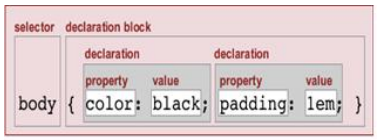

Figure 2: General CSS rule structure and a simple CSS rule example

class selectors, and attribute selectors among others (for more details, see [10]). To explain how the CSS rule-matching process works, we will use the sample HTML document tree shown in Figure 3 and the following sample rule : `ul em {color: blue}`. For the web document using this CSS rule (which is a descendant selector rule), any `<em>` element that is contained in a `<ul>` element (i.e., the `<ul>` element is an ancestor of the `<em>` element in the web document's tree) will be colored blue. This matching process is illustrated in Figure 3, with the `<em>` element that will potentially be colored blue being the hashed node in the figure and its ancestors being the lighter colored nodes in the highlighted path from the `<em>` element up to the root of the tree. The matching process is carried bottom up, starting with the parent of the `<em>` element, until either a match is found or the root of the document tree has been reached and no match has been found.

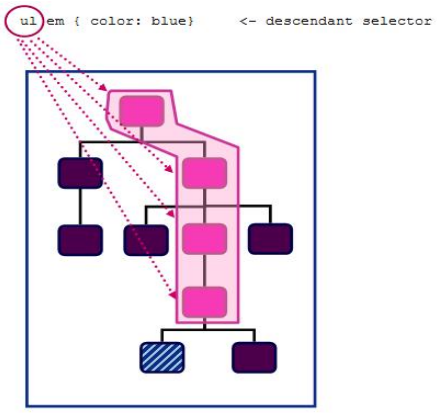

Figure 3: CSS rule-matching example on a sample HTML document tree

Since in the rest of the paper we will mostly refer to descendant and sibling selectors, we will briefly describe how they work.
A descendant selector is formed by two or more simple selectors that are separated by white space. The general form of such a selector is `'A B'` and it will match when an element of type `B` is a descendant-of/contained-in an element of type `A`.

Similarly, an (adjacent) sibling selector is made up of two or more selectors separated by `+`. For a rule of the form `A + B`, this selector will match if `A` and `B` have the same parent in the document tree and `A` immediately precedes `B`.

2.2 Profiling Strategy and Results

Before doing any parallelization, we needed to determine which Firefox component(s) accounted for large amounts of the browser execution time. According to our extensive VTune [1] experiments on many Intel platforms, the layout engine of Firefox is its heavy hitter performance component. For example, on Intel Atom platform, the layout component accounted for ∼40% of the total browser execution time, while the JavaScript execution engine covered only ∼13%. Further profiling showed that the CSS rule-matching process represents the hot part of the layout engine. The top two most executed functions of the layout engine, that account for an overall ∼32% of the execution time of the layout engine, were the `SelectorMatches` and `SelectorMatchesTree` functions, which are the most important components of the CSS rule-matching process. Unlike applications that are easily parallelizable (e.g., scientific/numerical applications), the rest of the execution time spent in the layout engine is thinly spread over a large number of components rather than concentrated in a small portion of code. Thus we focused our efforts to optimize the browser execution on the CSS rule-matching component as it had the best chances of making a considerable impact on the overall browser performance.

The rule-matching process involves executing a series of iterations for each selector in order to match the selector in question against the nodes of a tree-like, dynamic data structure that abstracts the underlying structure of each web page (see a very simple matching example outlined in Figure 3). This process is not only highly dynamic, but also highly speculative due to many decision points, which increases the difficulty of optimizing it.

In order to determine if the rule-matching process would benefit from parallelization and to assess the speedup potentials, we built a tracing mechanism based on VProf, the value-profiling package available in the Firefox source tree. We incorporated this tracing mechanism into the layout engine of Firefox and we conducted extensive experiments using the Zimbra Collaboration Suite performance tests [12], as well as the Firefox page-load benchmarks that were graciously supplied to us by the Mozilla Team (more details on these benchmarks in Section 3).

#Occurrences	Pattern	%iterations covered
5561595	p3	2.93
1997994	ppppppppppn1	13.43
1977238	ppppppppppn1	24.88
1846126	pn3	25.85
1599769	pppppppn1	32.58
1530957	ppppppppn1	39.83
1293911	pppppppppppn1	47.99
1187742	pppppppn1	52.37
840304	ppppppn1	55.02
619361	pppppppppppppn1	59.25
550482	ppppppppppppppn1	63.31
418940	pppppppppppppppn1	66.61
382272	ppppppppppppppppn1	69.83
310138	pppppn1	70.65

Table 1: Most frequent execution patterns observed during CSS rule matching on Firefox page-load tests

We found that in the CSS rule-matching process of typical web pages, not only are the selectors heavily biased towards one particular type (namely descendant selectors, which were seen 99% of the time for both benchmark suites), but also that in the vast majority of cases, the matching process for a CSS rule results in a non-match. In other words, the dominant execution pattern is carried out by a series of iterations over an underlying tree-like dynamic data structure that could be refactored to operate in parallel and independent of each other.

To illustrate the observations above, Table 1 shows the top 15 most frequent patterns of execution in the CSS rule-matching process for the Firefox page-load benchmarks. In this table, `'p'` indicates that a descendant selector match against an ancestor was attempted, while the `'1'` at the end of a pattern signals that the matching process ended in a non-match. As can be observed from Table 1, the rule matching is very heavily biased towards descendant selectors (if there were any sibling selectors involved, `'b'` symbols would be shown) and most rule matching attempts result in a non-match. Surprisingly, we observed a very similar tracing data for the Zimbra Collaboration Suite (ZCS) - a leading-edge AJAX-based, rich-browser integrated suite of email, calendar, contacts and VoIP.

Since rule matching accounts for a significant portion of execution time as discussed earlier, it is evident that parallelizing the CSS rule-matching component of Firefox, with a focus on non-matching descendant selectors based on the tracing data detailed above, should yield considerable performance speedups.

2.3 Parallel Implementation

With the profiling and tracing data analyzed in Section 2.2 in mind, we re-factored the CSS rule-matching algorithm of Firefox into a parallel equivalent. One should note that the sequential CSS rule-matching mechanism used in Firefox has been heavily optimized over years, making it even more challenging to achieve further performance improvements through parallelism. As we have described before in Section 2.2, the CSS rule-matching process involves applying a series of iterations for each selector in order to match the selector in question against the nodes of a tree-like, dynamic data structure that abstracts the underlying structure of each web page (see a very simple matching example outlined in Figure 3). Each iteration essentially consists of a `SelectorMatches` call that takes the current (ancestor) node being matched against as a parameter.

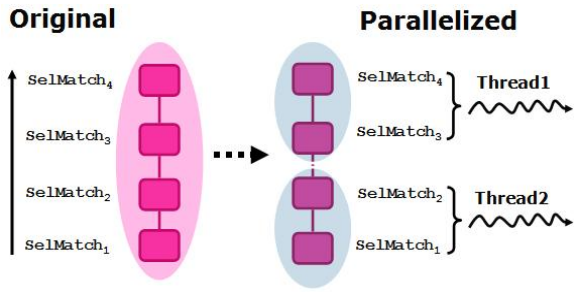

Figure 4: Transition from sequential to parallel rule matching for a sample CSS rule

Based on the empirical profiling and tracing results that we gathered, we utilized the observed two major properties of the CSS rule-matching process being heavily biased towards descendant selectors and the majority of CSS rule checks ending up in non-matches. Thus, in most of the cases, all the iterations that carry out the CSS rule-matching process are actually executed (i.e., selector matches need to be carried out against **all** ancestors of the selector node in question) and having them execute speculatively in parallel by an adjustable number of worker threads should lead to a considerable performance improvement.

Each thread is assigned a certain amount of ancestors to process. This amount is computed based on how many ancestors are available for the node under operation, as well as on the number of available worker threads. Figure 4 illustrates the transition from the sequential CSS rule-matching process to the parallelized version we developed, based on the CSS rule and document tree samples shown in Figure 3.

There is some degree of speculation involved, as not all iterations might have been executed in the original sequential version (i.e this corresponds to the case where there is a CSS rule match; however, as we outlined in Section 2.2, matches are quite rare); the speculative work that proves to be unnecessary is discarded. Furthermore, the code is also redesigned in such a way that the speculative work does not cause any unwanted side effects. There is also a check by each worker thread to avoid extraneous computation if another thread has already found a match.

The parallel CSS rule-matching process we implemented depends on three adjustable parameters:

- The number of worker threads available;
- The default workload size to be assigned to a worker thread for processing (i.e., the minimum number of selector matching iterations that a worker thread will execute);
- The threshold on the workload size that needs to be hit before enabling the parallel CSS rule matching. This is needed in order to amortize the overhead of thread management.

In the next section we will show the performance results obtained using the parallelized CSS rule-matching prototype we developed.

3 Performance Results

We experimented with a large number of configurations by varying the three parameters described in Section 2.3. Our measurements were performed on an single socket Quad-Core (2.93 GHz) Nehalem machine, with hyper-threading enabled and 2.49 GB RAM.

In the performance experiments with our parallelized CSS rule-matching prototype, we used two real-world benchmark suites:

- The Zimbra Collaboration Suite (ZCS) performance tests (described in Section 2.2).
- The Firefox page-load benchmarks that were provided to us by the Mozilla team. This is a collection of 390 webpages from all over the world that test the page loading process in Firefox.

Figure 5 shows the performance speedups obtained for the best performing parallel configuration on the Nehalem platform, using two worker threads for the parallelized layout component and running the Firefox page-load tests. As we can see, out of the 390 webpages tested, ~87% of them yield performance improvements, with speedups of up to a maximum of 1.84x.

Figure 6 shows the performance speedups obtained for the best performing parallel configuration on the

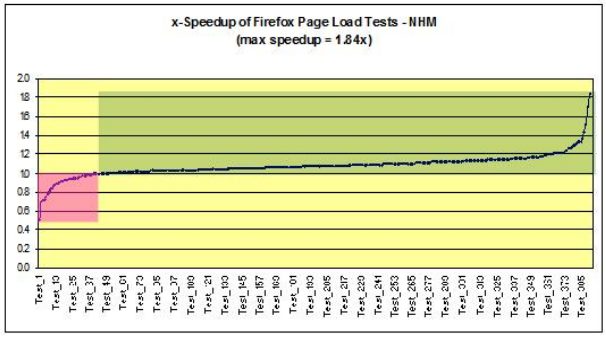

Figure 5: Performance speedups obtained on Intel Nehalem for the Firefox page-load tests

Nehalem platform, using two worker threads for the parallelized layout component and executing the Zimbra Collaboration Suite. As can be observed, out of all the ZCS benchmarks tested, ∼50% yield performance improvements, with speedups of up to a factor of 1.6, while ∼14% were not affected performance wise. The reason why the ZCS benchmarks

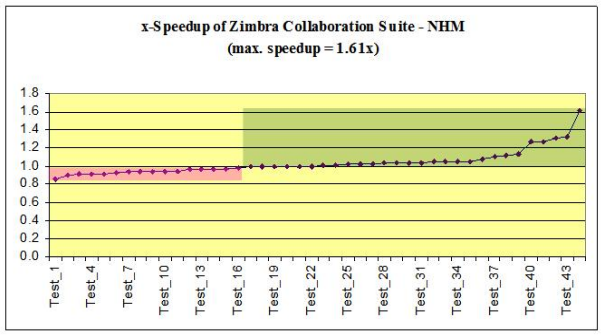

Figure 6: Performance speedups obtained on Intel Nehalem for the Zimbra Collaboration Suite

show overall less of a performance improvement compared to the Firefox page-load tests is because ZCS is a more JavaScript-oriented benchmark, as opposed to Mozilla's page load tests, which are used specifically to test layout features.

We experimented with varying the amount of worker threads available, but due to thread management overheads, in conjunction with the thread workload characteristics of the two benchmark suites we tested, we found that using two worker threads was the best option in terms of execution performance. Since the trend is towards more complex and more sophisticated web pages, we expect that the benefits from parallel layout would be on the rise, as more processing would be required for their layout.

As it can be observed from Figures 5 and 6, there are some tests for which we experience a certain amount of performance degradation. Based on the behavior observed during the development of our CSS rule-matching parallelized prototype, we believe that these instances correspond to cases where the workloads assigned to the worker threads are not large enough to mitigate the thread management overhead. We plan to address the slowdowns in our future work. It is worth noting that if we were to show the performance results obtained relative to the CSS rule-matching component alone, the speedups reported would have been much higher. However, we report the speedups relative to the overall browser performance since that is the measure that the end-user experiences.

4 Related Work

Web browsers are large, CPU-intensive and increasingly complex programs that present monolithic, single-threaded characteristics which are becoming unsuitable for the current multi-core trend observed both in general purpose systems, as well as in embedded devices. As mentioned in Section 1, the Google Chrome browser is adapting to the increasing multi-core presence by treating each tab as a separate process, thus naturally lending itself to multithreading.

In [7], Ha et al. propose a concurrent, trace-based JIT compiler prototype, incorporated in the Tamarin Tracing project [14] and showing throughput improvements ranging from 6% on average up to 34% when tested on the SunSpider benchmarks [3], thus being another good candidate for parallelizing the TraceMonkey JavaScript engine in the Firefox browser. Since the layout component currently accounts for a considerably larger portion of the execution time than the JavaScript engine (∼40% vs. ∼15%), we focused our efforts on parallelizing the Firefox layout engine.

Meyerovich and Bodik [8] present their work on designing a parallel web browser, namely parallel implementations of their own stand-alone CSS rule matching, layout solving, and font rendering. They report performance speedups of up to 80x. However, the CSS selector-matching implementation that is parallelized and tested for performance is based on their own browsers models, not based on the implementation in any of the finely-tuned and highly-optimized real browsers. As such, it is hard to compare these results to ours based on the layout performance of a popular world-class web browser.

King, et al. [6, 11] present their work on building a new web browser [9] to improve on the existing level of web browsing security. One feature in their browser implementation is parallelizing the operations of the window manager, which leads to per-

formance improvement for a few of the web pages tested [11]. The baseline in the performance experiments is the Arora browser [4], which is a free, lightweight cross-platform browser. Thus, it is hard to compare our results to theirs since our parallel prototype is based on a highly-optimized and rich-in-feature browser that is widely used (by far top among open-source browsers; see Figure 1).

5 Conclusions

We have developed a fully-functional parallel implementation of the CSS rule-matching process, based on the in-depth tracing and profiling data obtained indicating that the CSS rule-matching component is an excellent candidate for parallelization. We showed that it delivers large speedups when incorporated into the Firefox browser. On popular web pages, including almost 90% of the Mozilla's page-load benchmarks that comprise ∼400 distinct web pages from all over the world, our parallel layout implementation achieves up to 1.84x page-load speedup when two worker threads cooperate in the page layout. We have submitted our parallel CSS rule-matching implementation to Mozilla [13] and have received very encouraging feedback. We plan to further refine our parallelization strategy to address the small number of slowdowns that we have observed in some of the benchmarks.

6 Acknowledgments

The authors would like to thank the following people: David Baron, Brendan Eich, Chris Jones, Robert O'Callahan, and Boris Zbarsky of Mozilla for making Mozilla's page-load tests available and for their encouraging feedback and suggestions; Leo Meyerovich and Ras Bodik of the UC Berkeley for their inspiring work and constructive discussions; Andy Pflaum and Raja Rao of Yahoo/Zimbra for making the Zimbra Collaboration Suite and its performance tests available, and Geoff Lowney of Intel Corporation for his generous support of this project.

References

[1] VTune Performance Analyzer, http://software.intel.com/en-us/intel-vtune/.

[2] Browser Market Share as of December 2009, http://marketshare.hitslink.com/report.aspx?qprid=0.

[3] SunSpider Benchmarks, http://www2.webkit.org/perf/sunspider-0.9/sunspider.html.

[4] The Arora Browser, http://code.google.com/p/arora/.

[5] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz, *Trace-based Just-in-Time Type Specialization for Dynamic Languages*, PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation (New York, NY, USA), ACM, 2009, pp. 465–478.

[6] Chris Grier, Shuo Tang, and Samuel T. King, *Secure Web Browsing with the OP Web Browser*, SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy (Washington, DC, USA), IEEE Computer Society, 2008, pp. 402–416.

[7] Jungwoo Ha, Mohammad R. Haghighat, Shengnan Cong, and Kathryn S. McKinley, *A Concurrent Trace-based Just-In-Time Compiler for Single-threaded JavaScript*, 2009 Workshop on Parallel Execution of Sequential Programs on Multicore Architectures (PESPMA 2009), in conjunction with ISCA.

[8] Leo Meyerovich and Ras Bodik, *Fast and Parallel Webpage Layout*, WWW 2010: 19th International World Wide Web Conference (Raleigh, NC, USA), 2010.

[9] The OP Web Browser Source Code Repository, http://code.google.com/p/op-web-browser/.

[10] Cascading Style Sheets, http://www.w3.org/.

[11] Samuel T. King Shuo Tang, Chris Grier, *Making Secure Browsers Fast in the Multicore Era*, http://www.upcrc.illinois.edu/presentations/2009_09_17_Tang_Slides.pdf.

[12] The Zimbra Collaboration Suite, http://www.zimbra.com/.

[13] Bugzilla: Mozilla's Bug Tracking System, https://bugzilla.mozilla.org/.

[14] Tamarin Tracing, https://wiki.mozilla.org/Tamarin:Tracing.