# Parallel Performance-Energy Predictive Modeling of Browsers: Case Study of Servo

Rohit Zambre[⋆], Lars Bergstrom[†], Laleh Aghababaie Beni[*], Aparna Chandramowlishwaran[⋆]

[⋆]*EECS, University of California, Irvine, CA*
[*]*ICS, University of California, Irvine, CA*
[†]*Mozilla Research, USA*

***Abstract*—Mozilla Research is developing Servo, a parallel web browser engine, to exploit the benefits of parallelism and concurrency in the web rendering pipeline. Parallelization results in improved performance for *pinterest.com* but not for *google.com*. This is because the workload of a browser is dependent on the web page it is rendering. In many cases, the overhead of creating, deleting, and coordinating parallel work outweighs any of its benefits. In this paper, we model the relationship between web page primitives and a web browser's parallel performance using supervised learning. We discover a feature space that is representative of the parallelism available in a web page and characterize it using seven key features. Additionally, we consider energy usage trade-offs for different levels of performance improvements using automated labeling algorithms. Such a model allows us to predict the degree of parallelism available in a web page and decide whether or not to render a web page in parallel. This modeling is critical for improving the browser's performance and minimizing its energy usage. We evaluate our model by using Servo's layout stage as a case study. Experiments on a quad-core Intel Ivy Bridge (i7-3615QM) laptop show that we can improve performance and energy usage by up to 94.52% and 46.32% respectively on the 535 web pages considered in this study. Looking forward, we identify opportunities to apply this model to other stages of a browser's architecture as well as other performance- and energy-critical devices.**

## I. Introduction

For any particular browser, a heavier page takes longer to load than a lighter one [15]. The workload of a web browser is dependent on the web page it is rendering. Additionally, with the meteoric rise of the Web's popularity since the 1990s, web pages have become increasingly dynamic and graphically rich—their computational complexity is increasing. Hence, the page load times of web browsers have become a growing concern, especially when the user experience affects sales. A two-second delay in page load time during a transaction can result in abandonment rates of up to 87% [7]. Further challenging matters, an optimization that works well for one page may not work for another [29].

The concern of slow page load times is even more acute on mobile devices. Under the same wireless network, mobile devices load pages $3\times$ slower than desktops, often taking more than 10 seconds [22]. As a result, mobile developers deploy their applications using low-level native frameworks (*e.g.* Android, iOS, etc. applications) instead of the high-level browser, which is usually the case for laptop developers. However, these applications are hard to port to phones, tablets, and smart TVs, requiring the development and maintenance of a separate application for each platform. With a faster browser, the universal Web will become more viable for all platforms.

The web browser was not originally designed to handle the increased workload in today's web pages while still delivering a responsive, flicker-free experience for users. Neither was its core architecture designed to take advantage of the multi-core parallelism available in today's processors. An obvious way to solve the slow web browser problem is to build a parallel browser. Mozilla Research's Servo [14] is a new web browser engine designed to improve both memory safety, through its use of the Rust [13] programming language, and responsiveness, by increasing concurrency, with the goal of enabling parallelism in *all* parts of the web rendering pipeline.

Currently, Servo (see Section III) parallelizes its tasks for all web pages without considering their characteristics. However, if we naïvely attempt to parallelize web rendering tasks for all content, we will incur overheads from the use of excessive number of threads per web page. More importantly, we may also penalize very small workloads by increasing power usage or by delaying completion of tasks due to the overhead of coordinating parallel work. Thus, the challenge is to ensure fast and efficient page load times while preventing slowdowns caused by parallel overhead. We tackle this challenge by modeling, using accurate labels and supervised learning, the relationship between web page characteristics and the parallel performance of a web rendering engine and its energy usage within the *complete* execution of a browser. In this paper, we work with Servo since it is currently the only publicly available parallel browser. However, our modeling approach can easily extend to any parallel browser on any platform since our feature space is *blind* to the implementation of a web rendering engine.

Precisely, we model with seven web page features that represent the amount of parallelism available in the page. These features are oblivious to the implementation of a rendering engine. We correlate these features to the parallel performance in two stages of a parallel web rendering engine. The first stage, *styling*, is the process in which the engine determines the CSS styles that apply to the various HTML elements in a page. The second stage analyzed in this work is *layout*. During *layout*, the engine determines the final geometric positions of all of the HTML elements. We choose these two stages since

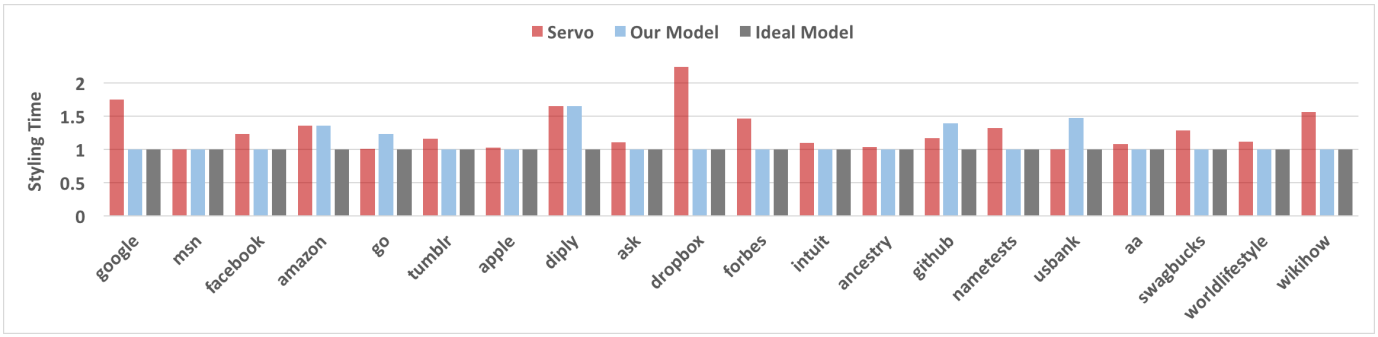



Fig. 1. Normalized styling times of Servo, Our Model, and Ideal Model.

they consume a significant portion of overall rendering time, especially for modern dynamic web pages. Internet Explorer and Safari spend 40-70% of their web page processing time, on an average, in the visual layout of the page [28].

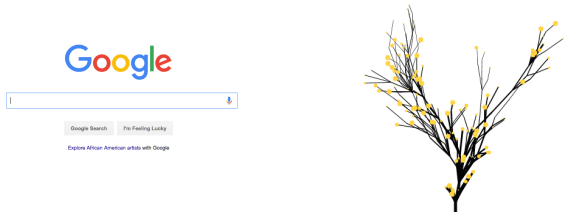
We evaluate our model for Servo on a quad-core Intel Ivy Bridge using off-the-shelf supervised learning methods on 535 web pages. Even with a large class-imbalance in our data set, we demonstrate strong accuracies, approaching 88%. Figure 1 depicts the styling times taken by Servo and our model against an optimal model (to which times are normalized) for the top 20 web pages in the Alexa Top 500 [3] list. *Ideal Model* represents the best time that is achieved using either 1, 2 or 4 threads (see Section IV for our experiment's configuration). *Our Model* represents the time taken using the number of threads suggested by our proposed model. *Servo* represents the time taken by 4 threads, the default number of threads that the Servo browser engine, unlike our model, spawns for the styling and layout stages on a quad-core processor. *Our Model* performs as well as the *Ideal Model* in most cases.

We make three main contributions:

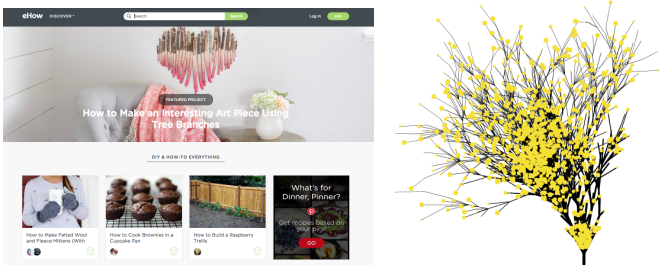
(1) **Workload characterization –** The workload of a browser is dependent on the web page. We study and analyze the Document Object Model (DOM) [5] tree characteristics of a web page and use them to characterize the parallel workload of the rendering engine (see Section II).
(2) **Performance-energy labeling of web pages –** Considering performance speedups and energy usage "greenups," [24] we label web pages into different categories using three cost models. To do so, we propose automated labeling algorithms for each cost model (see Section V).
(3) **Performance-energy modeling and prediction –** Using supervised learning, we construct, train, and evaluate our proposed statistical inference models that capture the relationship between web page characteristics and the rendering engine's performance and energy usage. Given the features of a web page, we use the model to answer two fundamental questions: (a) should we parallelize the styling and layout tasks for this web page? If so, (b) what is the degree of available parallelism? (see Section VI)

## II. Web Page Characterization

A wide variety of web pages exists in today's World Wide Web. Either a web page can contain minimal content with little to no images or text, or it can include a wide variety of multimedia content including images and videos. The left column of Figure 2 depicts the web pages of `google.com` and `ehow.com`, two contrasting instances that exemplify the variety of web pages that one comes across on a daily basis.



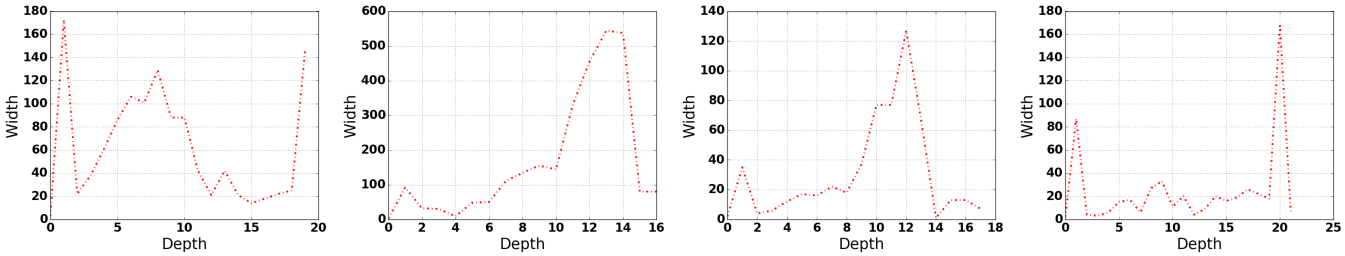

(a) `google.com`




(b) `ehow.com`

Fig. 2. Contrasting types of web pages (left) and the visualization of their DOM trees (right).

The right column of Figure 2 portrays a visual representation (created using Treeify [16]) of the DOM tree of the corresponding web pages. The DOM tree is an object representation of a web page's HTML markup. Qualitatively, Figure 2 shows that simple web pages, like `google.com`, have relatively small DOM trees, low number of leaves, and are not as wide or as deep. On the other hand, complex web pages, such as `ehow.com`, have relatively big trees, a high number of leaves, and are much wider and deeper.




(a) `airbnb.com` (b) `samsclub.com` (c) `westlake.com` (d) `facebook.com`

Fig. 3. Width Vs. Depth graphs of the DOM trees of different web pages (note that the scales of the axes are different).

Browser optimizations are primarily applied to the style application stage since it is the most CPU-intensive of all stages [6]. It is during this step that the DOM tree is traversed extensively to compute the styles for each element on the page. Naturally, a parallel browser would then optimize this stage using parallel tree traversals [28]. Hence, we identify DOM tree features that correlate strongly with the performance of these parallel tree traversals. Any amount of parallel speedup or slowdown would depend on the structure of the DOM tree. We intuitively choose the following set of nine characteristics to capture the properties of a web page and its DOM tree:

1. Total number of nodes in the DOM tree (**DOM-size**)
2. Total number of attributes in the HTML tags used to describe the web page (**attribute-count**)
3. Size of the web page's HTML in bytes (**web-page-size**)
4. Number of levels in the DOM tree (**tree-depth**)
5. Number of leaves in the tree (**number-of-leaves**)
6. Average number of nodes at each level of the tree (**avg-tree-width**)
7. Maximum number of nodes at a level of the tree (**max-tree-width**)
8. Ratio of max-tree-width to average-tree-width (**max-avg-width-ratio**)
9. Average number of nodes per level of the tree (**avg-work-per-level**)

Our intuition is that large* and wide trees observe higher speedups than small and narrow trees in parallel tree traversals (captured by **DOM-size** and **avg-tree-width**). In consecutive top-down and bottom-up parallel traversals (see Section III), trees with a large number of leaves observe faster total traversal completion time (captured by **number-of-leaves**). Even amongst wide trees, those that don't have abrupt changes in tree-width, or are less deep, observe faster parallel traversals (captured by **tree-depth**, **max-avg-width-ratio**). **DOM-size** captures the total amount of work while **avg-work-per-level** ($=$ **DOM-size**$/$**tree-depth**) captures the average parallel work on the web page. **attribute-count**, and **web-page-size** capture the general HTML information about a web page. Although we initially identify nine features, we only choose seven for modeling based on the results of the statistical correlation of these characteristics to Servo's parallel performance (see Section VI).

To quantitatively analyze DOM trees, we plot the width of the trees at each of their depth levels. In Figure 3, we do so for `airbnb.com`, `samsclub.com`, `westlake.com`, and `facebook.com`. Using these figures, we relate our intuition to observed data. `airbnb.com` and `samsclub.com` are examples of DOM tree structures that represent "good" levels of available parallelism. The **DOM-size** of `samsclub.com` is 2833 and hence, sufficient work is available. The **DOM-size** of `airbnb.com` is 1247 which is much smaller than that of `samsclub.com`. However, the DOM tree of `airbnb.com` has a high **avg-tree-width** of 62.3, a characteristic that favors parallelism. Our performance experiments show that `samsclub.com` achieves $1.48\times$ speedup with 2 threads and $2.12\times$ speedup with 4 threads. `airbnb.com` achieves speedups of $1.2\times$ and $1.43\times$ with 2 and 4 threads respectively, which, although significant, are not as high as those of `samsclub.com` due to the lesser amount of available work. The DOM trees of `westlake.com` and `facebook.com` exemplify tree structures that represent "bad" candidates for parallelism. These trees have large widths only for a small number of depth levels. Hence, the **avg-tree-width**s of these trees are low: 30.6 and 24.6 for `westlake.com` and `facebook.com` respectively. These trees don't have enough amount of work to keep multiple threads occupied. `westlake.com` shows slowdowns of $0.94\times$ and $0.74\times$ with 2 and 4 threads respectively. Similarly, `facebook.com` demonstrates slowdowns of $0.86\times$and $0.81\times$ with 2 and 4 threads respectively.

## III. Servo Overview

Servo [14] is a web browser engine that is being designed and developed by Mozilla Research. The goal of the Servo project is to create a browser architecture that employs inter- and intra-task parallelism while eliminating common sources of bugs and security vulnerabilities associated with incorrect memory management and data races. C++ is poorly suited to prevent these problems.

Servo is written in Rust [13], a new language designed by Mozilla Research specifically with Servo's requirements in mind. Rust provides a task-parallel infrastructure and a

*The values of the features lie on a continuous spectrum and so, we cannot assign discrete definition to descriptors such as "big," "large," "small," "wide," "narrow," etc.

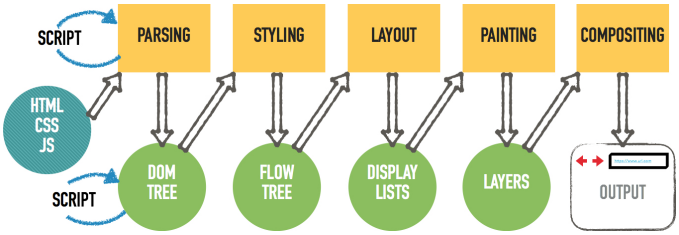



Fig. 4. Processing stages and intermediate representations in a browser engine. The circles represent data structures while the squares represent tasks.

strong type system that enforces memory safety and data-race freedom. The Servo project is creating both a full web browser, through the use of the purely HTML-based user interface BrowserHtml [4], and a solid embeddable web rendering engine. Although Servo was originally a research project, it was implemented with the goal of production-quality code and is in the process of shipping several of its components to the Firefox browser.

The processing steps used by all browsers are very similar, as many parts of the interpretation of web content are defined by the standards from the World Wide Web Consortium (W3C) and the Web Hypertext Application Technology Working Group (WHATWG). As such, the steps Servo uses in Figure 4 should be unsurprising to those familiar with the implementation of other modern browsers [6]. Due to space constraints, we describe only the phases relevant to this paper in detail. A more detailed description of these stages is available in [20], which, orthogonal to this work, outlines the language features of Rust in the context of Servo.

The first step in loading a site is retrieving and *parsing* the HTML and CSS files. The HTML translates into a DOM tree and the CSS loads into style structures. Each node of the tree corresponds to an HTML element in the markup. The CSS style structures are used in *styling*. JavaScript may execute twice, during parsing and after page-load during user-interactivity when it can modify the DOM tree.

### A. Styling

After constructing the DOM tree, Servo attaches styling information in the style structures to this tree. In this process, it builds another tree called the *flow tree* which describes the layout of the DOM elements on the page in the correct order. However, the flow tree and the DOM tree don't hold a one-to-one relation unlike that of the HTML markup and the DOM tree. For example, when a list item is styled to have an associated bullet, the bullet itself will be represented by a separate node in the flow tree even though it is not part of the DOM tree.

This stage contains the first step that is the subject of analysis in this paper. Servo executes styling using parallel tree traversals, an approach similar to the one employed by Meyerovich et al. [28]. Conceptually, the first half of this step is a trivially parallel process—each DOM tree node's style can be determined at the same time as any other node's. However, to prevent massive memory growth, Servo shares the concrete style structures that are associated with multiple

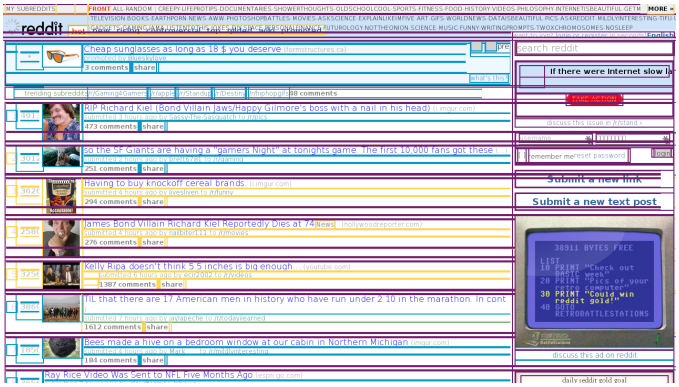


Fig. 5. Parallel layout on `reddit.com`. Different colors indicate that layout was performed by a different thread.

nodes, requiring communication between parallel threads. The second half of this step is the construction of the flow tree.

### B. Layout

The flow tree is then processed to determine the final geometric positions of all the elements first and then to produce a set of *display list* items.

Determining these positions is the second step that is analyzed in this paper. In cases where no HTML elements prevent simultaneous evaluation (*e.g.* floated elements and mixed-direction writing modes), Servo performs consecutive top-down and bottom-up parallel tree traversals to determine the final positions of elements; the height of a parent node is dependent on the cumulative height of its children, and the widths of the children are reliant on the width of the parent. These traversals execute incrementally and hence multiple individual passes occur before the end of a page-load. Figure 5 shows one parallel execution with four cores rendering `reddit.com`.

After final positions of the elements are computed, the engine constructs display list items. These list items are the actual graphical elements, text runs, etc. in their final on-screen positions. The order in which to display these items is well-defined by the CSS standard [17].

Finally, the to-be-displayed elements are *painted* into memory buffers or directly onto graphic-surfaces (*compositing*). Servo may paint each of these buffers in parallel.

In the rest of the paper, we will use the term **Overall Layout** to refer to the *Styling* and *Layout* stages together and **Primary Layout** to refer to the *Layout* stage alone. Also, unless specified otherwise, we refer to *total* times of all the incremental passes performed for each stage. The sequential baseline performance of Servo's *Overall Layout* is nearly 2× faster than Firefox's (Table 1 of [20]). This speedup stems from the use of Rust instead of C++ as more optimization opportunities exist in Rust. Additionally, the parallel implementation of *Overall Layout* has been optimized *i.e.* we observe high speedups on some websites: 3.2× and 2.5× with 4 threads on `humana.com` and `kohls.com` respectively. In some cases, 2 threads perform better than 4 threads: `walgreens.com` achieves 1.43× speedup with 2 threads

and $1.16\times$ with 4 threads. We attribute such slowdowns to the parallel overhead of synchronization, which we aim to mitigate with our modeling. Currently, Servo uses a work-stealing scheduler to schedule the threads that are spawned (once) to perform the parallel tree traversals in *Overall Layout*.

## IV. Experimental Setup

The Web is extremely flexible and dynamic. Constantly changing network conditions can add a significant amount of variability in any testing that involves acquiring data directly from the Internet. Further, due to advertising networks, A/B testing, and rapidly changing site content, even consecutive requests can have significantly different workloads.

Hence, to achieve repeatable and reliable performance results with Servo, we use Google's Web Page Replay [18] (WPR) to *record* and *replay* the HTTP content required for our tests. At a high level, WPR establishes local DNS and HTTP servers; the DNS server points to the local HTTP server. During *record*, the HTTP server acquires the requested content from the Internet and saves it to a local archive while serving the requesting client. During *replay*, the HTTP server serves all requests from the recorded archive. A 404 is served for any content that is not within the archive. We used WPR to *record* the web pages in our sample set first and then *replay* them during the experiments. The *replay* mode guarantees that Servo receives the same content every time it requests a particular web page. For our testing platform, we use a quad-core Intel i7-3615QM running OS X 10.9 with 8GB of RAM.

We collect our sample dataset from two sources: (1) Alexa Top 500 [3] web pages in the United States during January 2016, and (2) 2012 Fortune 1000 [1]. We initially started with $1000$ web pages but these contained domain names that were either outdated or corresponded to server names and not actual web pages (*e.g.* `blogspot.com`, `t.co`). Also, some web pages caused runtime errors when Servo tried to load them since it is a work in progress. After filtering out all such web pages, we have a working set of $535$ web pages.

For our performance testing, we use Servo's internal profiling tool that spits out a CSV file containing user times of the *Styling* and *Primary Layout* stages. For our energy testing, we use Apple's powermetrics [12] to capture processor power usage. In both the energy and timing experiments, we call Servo to open a web page and terminate it as soon as the *page load* is complete. Across all browsers, *page load* entails fully loading and parsing all resources, reflecting any changes through re-styling and layout. Servo goes a bit further in the automation harness and will also wait until any changes to the display list have been rendered into graphics buffers and until the in-view graphics buffers composite to the final surface. In the energy experiments, we allowed a sleep time of $10$ seconds before each run of Servo to prevent incorrect power measurements due to continuous processor usage.

Servo makes it possible to specify the number of threads to spawn in the *Overall Layout* stage. Given that our platform is a quad-core, we used $1$, $2$, and $4$ as the number of threads for each web page. Since Servo is still under development, we observe non-repeatable behavior. To account for repeatability, we run $5$ trials with each thread number for each web page for both the performance and energy experiments. With 5 trials, the medians of the *median absolute deviations* (MAD) of all 1-, 2- and 4-thread executions are low: $6.76$, $7.46$, and $7.49$ respectively. MAD is a robust measure of the variability of a data sample [26].

## V. Performance and Energy Automated Labeling

In this section, we propose tunable, automated labeling algorithms that can be used with any web browser on any testing platform to classify web pages into different categories. Automated labeling eliminates the need for a domain expert to manually and accurately label data. Given the labels, a predictive model can be trained using supervised learning methods. For labeling, we consider three cost models:

1. **Performance –** Labels depend only on performance improvements from parallelization.
2. **Energy –** Labels depend only on energy usage increases from parallelization.
3. **Performance and Energy –** Labels depend on both performance improvements and energy usage increases from parallelization.

Although we collected user times for both the *Styling* and *Primary Layout* stages on Servo, we focus our comparisons on the former for the following reasons – (1) The *Styling* stage works on the DOM tree while the *Primary Layout* stage works on the flow tree. Since we characterize web pages using the DOM tree, we expect *Styling* performance to correlate strongly to the tree characteristics. (2) The medians of total *Styling* time and total *Primary Layout* time as percentages of the *Overall Layout* time (single-thread execution) are $67.19\%$ and $7.83\%$. Clearly, *Styling* time primarily defines *Overall Layout* time. Figure 6 shows these percentages for a sample of $40$ randomly selected web pages.

Another interesting observation is that an individual serial pass of *Primary Layout* ranges between $1$ and $55$ ms. This range is much smaller than the $1$ to $320$ ms range of an individual serial *Styling* pass. Hence, parallelizing tree traversals for the *Primary Layout* stage will most likely result in poorer performance due to thread communication and scheduling overheads. Our results validate this analysis. On average, the time taken by an individual parallel pass for *Primary Layout* is $3.92$ ms, $7.01$ ms, and $9.82$ ms with 1, 2, and 4 threads respectively. We also observe an increase in the average total times for *Primary Layout* with parallelization: $221.39$ ms, $242.56$ ms, $263.73$ ms with 1, 2 and 4 threads respectively.

We compare the energy usage values of Servo in *Overall Layout* using 1, 2, and 4 threads. Although the data corresponds to the processor energy usage between the beginning and termination of Servo, these values are mainly affected by parallelization of *Overall Layout* because this stage constitutes the majority of the browser's execution time. We cannot obtain energy measurements at the granularity of function calls using Powermetrics [12], or any external energy profiling tool.

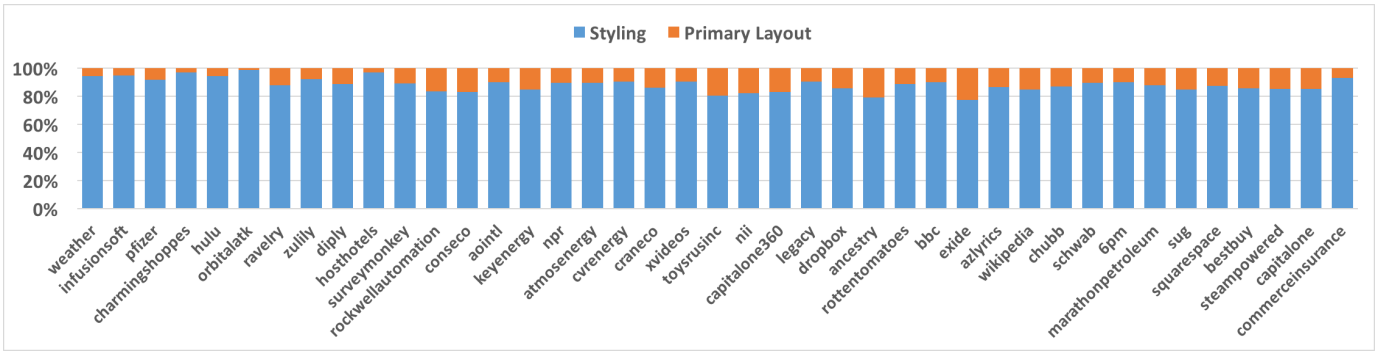



Fig. 6. Styling and Primary Layout time division in Overall Layout for a random sample of 40 web pages

### A. Performance Cost Model

In the Performance Cost Model, we consider only parallel runtimes to label the web pages appropriately. Consider an arbitrary number of thread configurations where each configuration uses $t$ threads. The values of $t$ are distinct. We first define the following terms.

- $x_t$ – time taken by $t$ threads
- $t_{\text{serial}} = 1$ (serial execution)
- $p_t = x_{t_{\text{serial}}}/x_t$ (speedup)
- $p_t^{\max}$ – maximum value of $p_t$
- $p_{\min}$ – minimum threshold that demarcates a significant speedup (to disregard measurement-noise)

The following steps describe the labeling process for a single web page:

1. For each thread configuration, we compute its speedup with respect to serial execution.
2. We calculate $p_t^{\max}$ for a web page using a maximum operation on the set of all its $p_t$ values.
3. If $p_t^{\max} > p_{\min}$, we assign a label $t$ where $t$ corresponds to that of $p_t^{\max}$. Otherwise, we label the web page as $t_{\text{serial}}$ since all other $p_t$ values would be smaller than $p_{\min}$.

**Algorithm 1** Performance labeling of a web page

```
Input:
T: { t | t is number of threads in a configuration }
P: { p_t | p_t is speedup using t threads, ∀ t ∈ T }
p_min: minimum threshold for significant speedup
procedure PERFORMANCE-LABELING
    p_t^max ← max(P)
    if p_t^max > p_min then
        label ← t
    else
        label ← t_serial
    return label
```

Hence, if there are $n$ thread-configurations, we have $n$ possible labels. If the label of a web page is $t$, it means that using $t$ threads achieves the best performance for that web page. Note that these labels are *nominal* values. They only identify the category and don't represent the total number of thread-configuration or their order. Algorithm 1 formally describes the classification of web pages using the Performance Cost Model.

For our experimental testbed with 4 cores, we had three thread-configurations: 1 thread ($t = 1$), 2 threads ($t = 2$), and 4 threads ($t = 4$). For a browser, where the running times are in the order of milliseconds, even a small performance improvement is significant. However, to account for noise in our measurements, we consider a $10\%$ speedup to be significant. We attribute speedups less than $10\%$ to noise. Hence, we set the threshold value of $p_{\min} = 1.1$. Using Algorithm 1, the total number of web pages categorized into labels 1, 2, and 4 are 299 $(55.88\%)$, 49 $(9.15\%)$, and 187 $(34.95\%)$ respectively.

### B. Energy Cost Model

In the Energy Cost Model, we consider only the energy usage values to label the web pages. The algorithm for labeling is the same as Algorithm 1. Instead of using speedup values, we consider greenup [24] (energy usage improvement) values. Let $y_t$ represent the energy consumed by $t$ threads. For each thread configuration, we compute its greenup, $e_t$ with respect to serial execution ($e_t = y_{t_{\text{serial}}}/y_t$). $e_{\min}$ is the minimum threshold that demarcates a significant greenup.

Since our experimental platform is a laptop, we did not use the Energy Cost Model to classify our web pages. We will consider this model in the future for energy-critical mobile devices.

### C. Performance and Energy Cost Model

In the Performance and Energy Cost Model, we consider both timing and energy usage values to label the web pages. In cases where we can guarantee significant performance improvements through parallelization, we also need to consider increases in energy usage. Spawning more threads could result in higher energy usage especially if the parallel work scheduler is a power-hungry one such as a work-stealing scheduler (as is the case currently in Servo). Hence, we consider performance improvements through parallelization to be useful only if the corresponding energy usage is lesser than an assigned upper limit. We label web pages using this cost model with a bucketing strategy as described below.

Similar to the classification in the previous two cost models, we consider an arbitrary number of thread configurations where each configuration uses $t$ threads. Each thread configuration has a corresponding speedup, $p_t$ and a greenup, $e_t$. In addition to the terminology defined in the previous two subsections, we define the following terms.

- $PET_t$ – *performance-energy tuple (PET)*, $\{p_t, e_t\}$ which represents the speedup and greenup achieved using $t$ threads.
- $P_jP_{j+1}$ – *PET bucket* to which a certain number of PETs belong. $P_j$ and $P_{j+1}$ represent speedup values where $j \in \mathbb{N}$. A PET, $PET_t \in P_jP_{j+1}$ if $P_j < p_t < P_{j+1}$. One can define an arbitrary number of such buckets to categorize the tuples. Note that the value of $P_1$ (lower limit of the first bucket) is always $p_{\text{min}}$.
- $E_j$ – energy usage increase limit (defined in terms of greenup) for a performance bucket $P_jP_{j+1}$ where $j \in \mathbb{N}$. $E_j$ demarcates the tolerance of energy usage increase for all $PET_t \in P_jP_{j+1}$.

In this labeling, we perform the following steps for each web page:

1. We ignore all the values of $p_t$ that are lower than $p_{\text{min}}$ and we define PET buckets based on design considerations.
2. If the filtering results in an empty set, we label the web page as $t_{\text{serial}}$. Otherwise, we organize the remaining speedups and greenups into PETs and assign them to the right PET buckets.
3. Starting from the last bucket (one with highest speedups),
   a) We sort the PETs in the descending order w.r.t. $p_t$ values.
   b) We look at the PET with the highest speedup, $p_t$ within this bucket and check to see if the corresponding energy usage, $e_t$ is less than the bucket's energy usage limit, $E_j$. If the check is not satisfied, we look at the next largest speedup in this bucket and repeat this step.
   c) When all PETs in a bucket don't satisfy the condition, we look at a lower bucket (one with the next highest speedups) and repeat the process. We do so until a PET satisfies the check against the energy usage limit.
4. If none of the PETs satisfy the condition, we label the web page as $t_{\text{serial}}$. Otherwise, we label the web page as the value of $t$ corresponding to the first PET that satisfies the condition.

Algorithm 2 formally describes the classification of the web pages using the Performance and Energy Cost Model and Figure 8 portrays a visual representation of the same.

For our case study with Servo, we had three thread configurations: 1 thread ($t = 1$), 2 threads ($t = 2$), and 4 threads ($t = 4$). We set the value of $p_{\text{min}} = 1.1$ (from Section V-A). Figure 7 depicts the histogram of Servo's $p_2$ and $p_4$ values. The histogram shows that, out of the significant speedups (~40%), the half point lies roughly at 1.3. Thus, we used two performance buckets: $P_1P_2$ and $P_2P_3$ where $P_1 = p_{\text{min}}$, $P_2 = 1.3$, and $P_3 = 12.87$ (the largest observed speedup). For the first bucket, we set the energy usage increase tolerance,

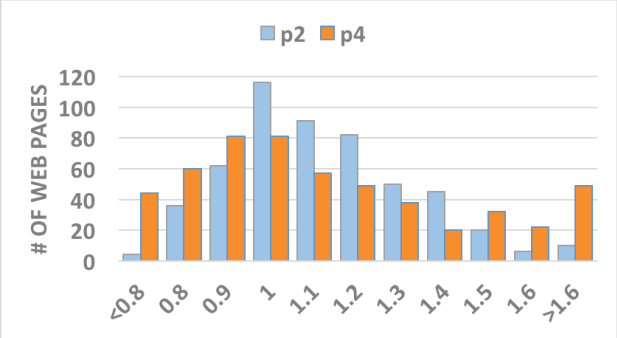



Fig. 7. Histogram of speedup values. The bin labels are upper limits.

$E_1 = 0.9^{\dagger}$ since a 10% energy usage increase can make a noticeable difference in overall battery life of a laptop. For the second bucket, we chose a tolerance, $E_2 = 0.85$ since an energy usage increase beyond 15% is not acceptable for any performance improvement. Using Algorithm 2, the total number of web pages categorized into labels 1, 2, and 4 are 317 (59.25%), 50 (9.34%), and 168 (31.40%) respectively.

**Algorithm 2** Performance-Energy labeling of a web page

**Input:**
$T$: $\{ t \mid t$ is number of threads in a configuration $\}$
$PET$: $\{ PET_t \mid PET_t = \{p_t, e_t\},$ where
$p_t$ is speedup using $t$ threads,
$e_t$ is greenup using $t$ threads, $\forall t \in T \}$
$P$: $\{ P_jP_{j+1} \mid P_jP_{j+1}$ is a bucket of PETs whose $P_j < p_t < P_{j+1} \}$
$E$: $\{ E_j \mid E_j$ is energy usage increase limit for $P_jP_{j+1}$, $\forall P_jP_{j+1} \in P \}$
$p_{\text{min}}$: minimum threshold for significant speedup
**procedure** PERFORMANCE-ENERGY-LABELING
    $label \leftarrow t_{\text{serial}}$
    **for** $P_jP_{j+1} \in P$ **do** // highest $j$ to lowest $j$
        $PET^{'} \leftarrow$ all $PET_t \in P_jP_{j+1}$
        $PET^{''} \leftarrow sortDescending(PET^{'}$ w.r.t $p_t)$
        **for each** $PET_t \in PET^{''}$ **do**
            **if** $e_t > E_j$ **then**
                $label \leftarrow t$
                break
    **return** label

## VI. Performance Modeling and Prediction

Our aim is to model the relationship between a web page's characteristics and the parallel performance of the web rendering engine to perform styling and layout on that page. With such a model, given a new web page, a browser will be able to predict the parallel performance improvement. The browser can then decide the number of threads to spawn during *Overall Layout* for a given web page using a statistically constructed model. When parallelization is beneficial, the browser should also consider energy usage values and check for tolerable

$^{\dagger}$These values can be tweaked based on design and device considerations. We choose these values for Servo on a quad-core Intel Ivy Bridge.

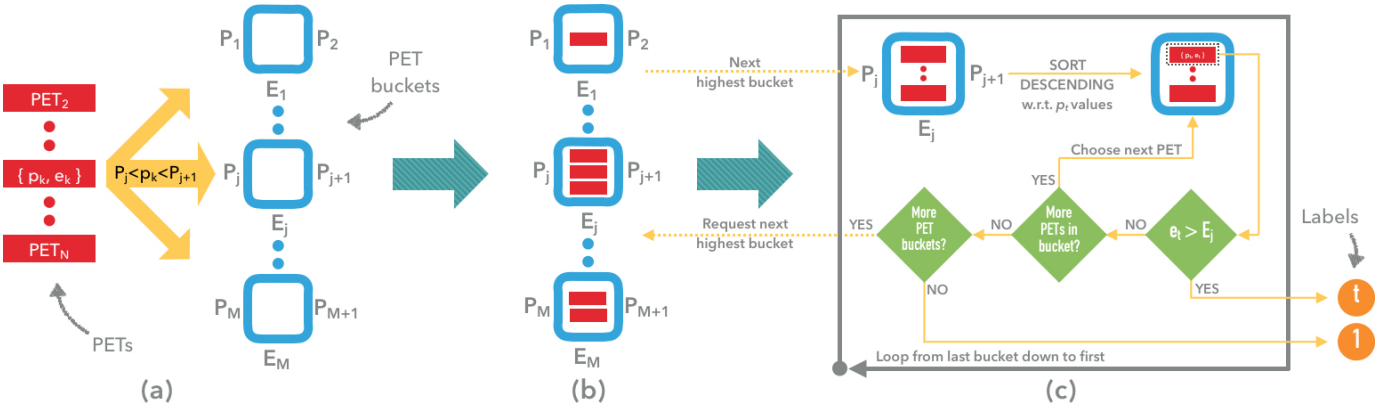



Fig. 8. A visual representation of the Performance and Energy Cost Model's labeling algorithm for a given web page. (a) $N$ PETs and $M$ PET buckets. (b) Based on $P_j$ and $P_{j+1}$ values, the PETs are assigned to the right buckets. (c) Algorithm-flow to choose the correct label for a web page.

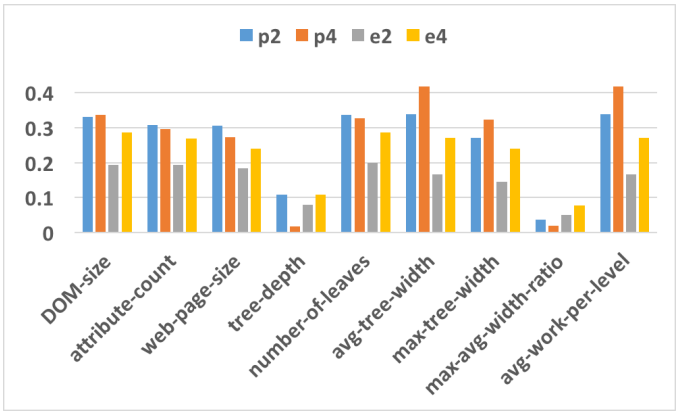



Fig. 9. Statistical correlation strengths: $R$-values.

amounts. Hence, our goal is to build a predictive model that allows a browser to decide the number of threads to spawn for its *Overall Layout* stage for any given web page by only looking at the web page's essential characteristics.

In this section, we construct and describe predictive models for Servo using off-the-shelf supervised learning methods on the Performance and Energy Cost Model (see Section V-C). For each of the models, our predictive features are the characteristics of a web page that we describe in Section II. Since the features lie on different scales[‡], we use Normal Standardization[§] (using MATLAB's `zscore` [11]) on the features to prevent one feature to dominate the other during learning. Figure 9 shows the correlation coefficient, $R$-values between the nine features and the speedups and greenups observed with 2 ($p_2$, $e_2$ values) and 4 ($p_4$, $e_4$ values) threads. We see that a positive linear[¶] relation exists between parallel benefits and the web page features. However, the exact relationship between the predictors and results is unlikely to be strictly linear, as is evident from our modeling results. Also, we see that **tree-depth** and **max-avg-width-ratio**, by themselves, do not correlate strongly ($< 0.1$) with the speedups and greenups observed in our dataset. However, when used in combination with other features, as in **avg-work-per-level**, we note a stronger relationship. For our modeling, we choose the seven predictor features that have $R$-values greater than $0.1$. The output for each of the models is one of the three labels: 1, 2, or 4 representing 1 thread, 2 threads, and 4 threads respectively. These response labels are *nominal* values.

Our dataset has a *large* class imbalance *i.e.* one label has many more observations than the other. For the Performance and Energy Cost Model, only 49 instances are labeled 2 while 299 and 187 are labeled 1 and 4 respectively. Also, the predictor to observation ratio for our data set is quite small. Hence, we don't face the issue of over-fitting.

To validate and test our models, we use *cross-validation* using a 90-10% training-testing ratio *i.e.* we divide our dataset into 10 subsets. Using each of these 10 subsets as the testing set (55 samples) and the remaining data as the training set (480 samples), we train 10 models. For each of the models, we predict labels for their corresponding testing sets, which results in 10 model prediction accuracies. We consider the mean and maximum of these 10 accuracies. When we are not able to use cross-validation, we use the *holdout-set* technique (again using a 90-10% training-testing ratio) wherein the data is divided into two sets: training and testing. We choose to use the cross-validation or holdout-set techniques because they result in true prediction error values and not "model errors" [2]. Additionally, they don't rely on any parametric or theoretic assumptions about the data.

We experimented with the following supervised learning algorithms: Multinomial Logistic Regression (MNR), Ensemble Learning, and Neural Networks. We choose these three methods to capture non-linear relationships between web page characteristics and a rendering engine's parallel performance.

**MNR**: In this multinomial logit model, the probability of each label for a web page is expressed as a non-linear function, using logit link functions of the predictor tree characteristics. We trained our model using MATLAB's `mnrfit` [9] function. With a 90-10% training-test divide for cross-validation, we observe a mean and maximum accuracy of 72.22% and

[‡]*e.g.* **DOM-size**'s statistical range is 8307 while **avg-tree-width**'s is 436.

[§]The values of a feature are distributed according to its Normal distribution such that the mean of the feature is 0 and its standard deviation is 1.

[¶]Statistical correlation measures the strength of only linear relationships between two variables.

87.27% respectively.

**Ensemble Learning**: For Ensemble Learning framework, the results of many models are combined to generate the final prediction. For our data, we used the AdaBoostM2 (a variant of Adaptive Boosting for multi-class data) learning method on 100 simple Decision Tree learners and also on 100 Decision Trees with surrogate splits. We did so using MATLAB's `fitensemble` [8] function. Using regular trees, with a 90-10% divide, we observe a mean accuracy of 71.12% and a maximum of 83.63%. Using trees with surrogate splits, we see a mean accuracy of 69.44% and a maximum of 85.45%.

**Neural Networks**. Artificial Neural Networks consist of simple, connected elements. By training the weights of the connections between different elements, a large neural network can capture complex relationships between variables. For our data, we use a small neural network with 1 hidden layer containing 10 neurons. We do so using MATLAB's `nprtool` [10]. Since MATLAB currently does not support cross-validation for its classifying neural network models, we use the holdout-set method to measure the accuracy of this model. We use 80% of the data for training, 10% for validation (which is essentially a part of training), and 10% for testing. The accuracy of this model on the testing set is 77.8%; the accuracy on all of the data is 71.61%. We attribute these lower (than those of MNR and Ensemble Learning) accuracies to the simplicity of the model and the lack of cross-validation.

The high accuracies of these off-the-shelf learning methods emphasize the effectiveness of our automated labeling algorithms. Instead of concentrating our efforts on tweaking the parameters of machine learning techniques to extract higher accuracies on our limited dataset, we demonstrate high accuracies by using *accurate* labeling algorithms that can be used on any dataset.

**Limitations**. These accuracies, however, are not greater than 90%. Machine learning models have the potential to be highly accurate but are heavily dependent on a large amount of accurate training data. They behave as "black boxes" with the actual underlying relationships between variables remaining undiscovered to the user [19], [32], [33]. Our data for this case study is relatively small and is from a prototype; Servo is a project under development and is undergoing constant change. Many components of Servo haven't been optimized for performance yet. Also, with multiple threads being spawned to exploit and explore concurrency in browser tasks, repeatability of executions is hard to acquire. Consequently, our data has a fair share of outliers—35% of the working set of web pages observes a MAD greater than 25. The class imbalance in our dataset is also an important factor that influences model accuracies since we don't have an equal share of training data for each class.

Despite our limitations, we observe *confident* accuracies on each of these models. This exactitude fosters our intuition that the web page characteristics are indeed related to the parallel performance and energy usage of a browser and that the practical parallelism benefits are predictable by these features. The best performing model, MNR, compared to the current implementation of Servo, achieves a maximum of 94.52% performance savings (2.48 ms with 1 thread vs. 45.41 ms with 4 threads on `indeed.com`) and a maximum of 46.32% energy savings (84.88 J with 1 thread vs. 158.14 J with 4 threads on `starbucks.com`). By performance savings of $x$%, we mean our model shaves off $x$% of the program's execution time. Similarly, by energy savings of $y$%, we mean our model shaves off $y$% of the program's energy usage.

## VII. Related Work

Research on parallel browser architectures and browser tasks began only recently, starting, to the best of our knowledge, in 2009. Although multi-core processors are ubiquitous today on both laptops and mobile devices, the browsers are yet to utilize their benefits. The growing concern of slow page load times, especially on mobile devices, and the unexploited parallelism benefits in commodity browsers are the primary motivations for this ongoing research. Below we outline existing research on parallelizing and analyzing browser-tasks.

**Browser Workload Characterization**. Gutierrez et al. [25] present characterization of an Android browser at the micro-architecture level using 11 web pages. Our approach, on the other hand, is agnostic to the platform on which a browser runs. Zhu et al. [35] correlate the web page variances to the difference in page load times and energy usage of the *serial* Firefox browser by characterizing the HTML and CSS elements of a web page. Our approach is similar in spirit: we consider only web page features to be predictive of a browser's performance. However, we find additional DOM tree features that are representative of the degree of the *parallel* workload in a page.

**Browser Performance-Energy Analyses**. Thiagarajan et al. [30] present a breakdown of energy usage by the different elements, such as CSS and Javascript, of a serial browser. Additionally, they propose a few optimizations to improve the power consumption of web browsers, such as re-organizing Javascript files or removing unnecessary CSS rules. Zhu et al. also focus on scheduling methods of heterogeneous systems to improve the energy efficiency of mobile processors. They evaluate benefits of a big.LITTLE heterogeneous system for a trade-off between performance and energy. However, both these projects assess on serial browsers. Our work, on the other hand, analyzes performance and energy trade-offs for a *parallel* browser while remaining agnostic to the browser implementation and execution platform. To the best of our knowledge, this work is the first of its kind.

**Parallel Browsers**. In general, browsers use processes to isolate tabs and windows to enhance security. Gazelle [31] uses two processes per page while Chrome uses a process-per-page approach. This method doesn't exploit parallelism within the tasks. ZOOMM [23] explores the challenges in managing concurrency in multi-core mobile-device browsers. This work exploits parallelism within the styling, image decoding, and JavaScript tasks leaving layout as future work. Mai et al. [27] propose that browser developers should focus on parallelizing web pages rather than browser-tasks. They build Adrenaline,

a server-client browser. On the other hand, Mozilla Research's Servo exploits both safety, particularly through its use of Rust, and parallelism between browser tasks and also within the tasks themselves. So far, Servo has exploited parallelism in its styling, layout and painting tasks, and is continuing to explore parallelism in other computational bottlenecks such as parsing. Servo is being developed for Android as well. More importantly, our work is not on parallelizing layout but is in predicting the degree of parallelism inherent in rendering a web page by considering the parallel performance and energy usage of a browser.

**Parallelizing Browser Tasks**. Several research projects on parallelizing browser-tasks such as styling [21], [28] and parsing [34] exist. Meyerovich et al. [28] introduce fast and parallel algorithms for CSS selector matching, layout solving and font rendering, and demonstrate speedups as high as $80\times$ using $16$ threads for six websites. However, they implement only a subset of CSS (without cascading) and evaluate their algorithm in isolation (not within a browser) without considering effects on energy usage. Servo adheres to the complete CSS specification, and our work is modeled within the *complete* execution of a browser while considering energy usage constraints.

## VIII. Conclusion

The workload of a web rendering engine is dependent on the web page it is rendering; we model the relationship between key web page features and the parallel performance of the rendering engine using supervised learning methods. Specifically, we characterize web pages using DOM tree and HTML characteristics that correlate to the *Overall Layout* task but are blind to the rendering engine's implementation. We propose accurate and tunable, automated labeling algorithms that categorize web pages into a user-defined number of classes. Moreover, our algorithm accounts for trade-offs between performance improvements and energy usage increases for multi-core processors. Using multinomial logit classification, ensemble learning, and simple neural networks, we demonstrate robust predictive model accuracies, achieving $87.27\%$ with 535 web pages within the complete execution of a browser. On a laptop platform, our best performing model delivers performance and energy savings up to $94.52\%$ and $46.32\%$ respectively.

## Acknowledgments

We would like to thank Sean McArthur from Mozilla, Subramanian Meenakshi Sundaram and Forough Arabshahi from the University of California, Irvine and others from Mozilla Research for helping us in conducting and analyzing our experiments.

## References

[1] 2012 Fortune 1000. http://booleanstrings.com/wp-content/uploads/2014/01/fortune1000-2012.xls.
[2] Accurately Measuring Model Prediction Error. http://scott.fortmann-roe.com/docs/MeasuringError.html.
[3] Alexa top 500. http://www.alexa.com/topsites/countries/US.
[4] BrowserHTML. https://github.com/browserhtml/browserhtml.
[5] DOM definition. http://www.w3.org/DOM/#what.
[6] How browsers work. http://www.html5rocks.com/en/tutorials/internals/howbrowserswork/.
[7] How Fast Should A Website Load? http://www.hobo-web.co.uk/your-website-design-should-load-in-4-seconds/.
[8] MATLAB fitensemble. http://www.mathworks.com/help/stats/fitensemble.html.
[9] MATLAB mnrfit. http://www.mathworks.com/help/stats/mnrfit.html.
[10] MATLAB nprtool. http://www.mathworks.com/help/nnet/ref/nprtool.html.
[11] MATLAB zscore. http://www.mathworks.com/help/stats/zscore.html.
[12] Powermetrics. https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man1/powermetrics.1.html.
[13] The Rust language. http://www.rust-lang.org/.
[14] The Servo web browser engine. https://github.com/servo/servo.
[15] Study: Load Times For 69% Of Responsive Design Mobile Sites Deemed Unacceptable. http://marketingland.com/study-load-time-69-mobile-sites-deemed-unacceptable-81126.
[16] Treeify. http://treeify.herokuapp.com/.
[17] W3 painting order. http://www.w3.org/TR/CSS21/zindex.html#painting-order.
[18] Web Page Replay. https://github.com/chromium/web-page-replay.
[19] E. Alpaydin. *Introduction to machine learning*. MIT press, 2010.
[20] B. Anderson, L. Bergstrom, M. Goregaokar, J. Matthews, K. McAllister, J. Moffitt, and S. Sapin. Engineering the Servo Web Browser Engine using Rust. In *Proceedings of the International Conference on Software Engineering 2016*, ICSE '16, New York, NY, USA, 2016. ACM.
[21] C. Badea, M. R. Haghighat, A. Nicolau, and A. V. Veidenbaum. Towards Parallelizing the Layout Engine of Firefox. In *Proc. of the 2nd USENIX Conf. on Hot topics in parallelism*, pages 1–1. USENIX Assoc., 2010.
[22] M. Butkiewicz, D. Wang, Z. Wu, H. V. Madhyastha, and V. Sekar. Klotski: Reprioritizing Web Content to Improve User Experience on Mobile Devices. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 439–453, 2015.
[23] C. Cascaval, S. Fowler, P. Montesinos-Ortego, W. Piekarski, M. Reshadi, B. Robatmili, M. Weber, and V. Bhavsar. ZOOMM: A Parallel Web Browser Engine for Multicore Mobile Devices. In *Proc. of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Prog.*, PPoPP '13, pages 271–280, New York, NY, USA, 2013. ACM.
[24] J. Choi, D. Bedard, R. Fowler, and R. Vuduc. A roofline model of energy. In *Parallel and Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 661–672. IEEE, 2013.
[25] A. Gutierrez, R. G. Dreslinski, T. F. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver. Full-System Analysis and Characterization of Interactive Smartphone Applications. In *Workload Characterization (IISWC), 2011 IEEE Intl. Symposium on*, pages 81–90. IEEE, 2011.
[26] C. Leys, C. Ley, O. Klein, P. Bernard, and L. Licata. Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median. *Jrnl. of Exp. Social Psyc.*, 49(4):764–766, 2013.
[27] H. Mai, S. Tang, S. T. King, C. Cascaval, and P. Montesinos. A Case for Parallelizing Web Pages. In *Presented as part of the 4th USENIX Workshop on Hot Topics in Parallelism*, 2012.
[28] L. A. Meyerovich and R. Bodik. Fast and Parallel Webpage Layout. In *Proc. of the 19th Intl. Conf. on WWW*, pages 711–720. ACM, 2010.
[29] J. Nejati and A. Balasubramanian. An In-depth study of Mobile Browser Performance. In *Proc. of the 25th Intl. Conf. on WWW*, pages 1305–1315. Intl. WWW Conf. Steering Committee, 2016.
[30] N. Thiagarajan, G. Aggarwal, A. Nicoara, D. Boneh, and J. P. Singh. Who Killed My Battery?: Analyzing Mobile Browser Energy Consumption. In *Proc. of the 21st Intl. Conf. on WWW*, pages 41–50. ACM, 2012.
[31] 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 *USENIX security symposium*, volume 28, 2009.
[32] K. Warwick. *March of the machines: the breakthrough in artificial intelligence*. University of Illinois Press, 2004.
[33] K. Warwick. *Artificial intelligence: the basics*. Routledge, 2012.
[34] Z. Zhao, M. Bebenita, D. Herman, J. Sun, and X. Shen. HPar: A practical parallel parser for HTML–taming HTML complexities for parallel parsing. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(4):44, 2013.
[35] Y. Zhu and V. J. Reddi. High-Performance and Energy-Efficient Mobile Web Browsing on Big/Little Systems. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 13–24. IEEE, 2013.