UNIVERSITY OF CALIFORNIA,
IRVINE

Adaptive Parallelism in Browsers

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Computer Engineering

by

Rohit Zambre

Thesis Committee:
Assistant Professor Aparna Chandramowlishwaran, Chair
Associate Professor Athina Markopoulou
Associate Professor Charless Fowlkes

2017

DEDICATION

To my accidental encounter with Aparna.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

I would like to thank Dr. Aparna Chandramowlishwaran for being a visionary motivator for this project. Her input, advise and push for excellence brought out the results portrayed in this thesis.

I would like to thank Mozilla Research for their continued technical and financial support throughout the entirety of this project. In particular, I would like to thank Dr. Lars Bergstrom for his flexible availability, invalueable guidance, and help.

I would like to thank Dr. Charless Fowlkes for inspiring new perspectives of looking at the problem-at-hand. I would like to thank Dr. Athina Markopoulou for her input, time and attention towards this project.

I would like to thank my labmate Laleh Aghababaie Beni for laying the foundations of our collaboration with Mozilla Research and for successful preliminary work that led to this thesis.

This work wouldn't have been possible without the discussions I had with my friends and with Servo-contributors on the #servo irc-channel. In particular, I would like to thank Subramanian Meenakshi Sundaram, Forough Arabshahi and Korosh Vatanparvar from the University of California, Irvine, and Sean McArthur from Mozilla Research.

I would like to thank IEEE for permitting me to include the content my work, titled "Parallel Performance-Energy Predictive Modeling of Browsers: Case Study of Servo," that was published in the 23rd IEEE International Conference on High Performance Computing, Data, and Analytics.

ABSTRACT OF THE THESIS

Adaptive Parallelism in Browsers

By

Rohit Zambre

Master of Science in Computer Engineering

University of California, Irvine, 2017

Assistant Professor Aparna Chandramowlishwaran, Chair

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 work, I model the relationship between web page primitives and a web browser's parallel performance and energy usage using both regression and classification learning algorithms. I propose a feature space that is representative of the parallelism available in a web page and characterize it using seven key features. After training the models to minimize custom-defined loss functions, such a model can be used to predict the degree of parallelism available in a web page and decide the optimal thread configuration to use to render a web page. Such modeling is critical for improving the browser's performance and minimizing its energy usage. As a case study, I evaluate the models on Servo's styling stage. 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 tackle this problem with an online-learning approach to realize a practical and portable adaptive parallel browser on various performance- and energy-critical devices.

Chapter 1

Introduction

For any browser, a heavier page takes longer to load than a lighter one [24]. The workload of a web browser is dependent on the web page it is rendering. 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% [11]. Further challenging matters, an optimization that works well for one page may not work for another [42].

The concern of slow page load times is even more acute on mobile devices. Under the same wireless network, mobile devices load pages 3× slower than desktops, often taking more than 10 seconds [34]. 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. One way to solve the slow web browser problem is to build a parallel browser in order to tap the multi-core prowess that is ubiquitous today—even the Apple Watch houses a dual-core processor. Parallelism in browsers is, however, not a new concept. Not only have researchers designed new, parallel algorithms for the computationally intensive tasks of a browser but they also have conducted parallel performance limit-studies. Meyerovich et al. [40] introduce fast and parallel algorithms for CSS selector matching, layout solving and font rendering, and demonstrate speedups as high as 80× using 16 threads for six websites; the offline analysis of Fortuna et al. [36] demonstrates that we can potentially achieve an average speedup of 8.9× by parallelizing JavaScript functions. Section 8 delineates the research conducted in this domain.

Motivated by multiple successes in parallelizing browser tasks, Mozilla Research embarked on the journey of designing and developing a new, parallel browser engine: Servo. Servo [23] is a new web browser engine designed to improve both memory safety, through its use of the Rust [21] programming language, and responsiveness, by increasing concurrency, with the goal of enabling parallelism in *all* parts of the web rendering pipeline.

Currently, Servo (Section 2) 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. I tackle this challenge by modeling 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. I work with Servo since it is currently the only publicly available parallel browser. However, the supervised learning modeling approach can easily extend to any parallel browser on any platform since the feature space is *blind* to the implementation of a web rendering engine.

Precisely, I model with eight 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 the most computationally intensive stage of a parallel web rendering engine: *styling*. Styling is the process in which the engine determines the CSS styles that apply to the various HTML elements in a page. This stage consumes 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 [40].

Using the eight predictive features, I define models using two approaches: regression and classification. These two approaches fit well to relationships between the features and the output variables observed in the dataset (Section 6). In the regression approach, a linear combination of the eight features predicts the performance and energy usage of each thread-configuration and then an automated decision-making algorithm chooses the best one. In the classification approach, a linear combination of the features spits out probabilities that the feature-sample belongs to one of the thread configurations. The thread configuration with the maximum probability is predicted as the best one. The automated decision-making algorithm is used here for the purpose of labeling.

After collecting performance and energy data for Servo on a quad-core Intel Ivy Bridge, I optimize the models over custom loss functions designed for each of the modeling approaches. I evaluate the models and *Servo* against an *Optimal Model* that always chooses the optimal thread configuration *i.e.* the one that maximizes performance while considering energy us-

age. *Servo* blindly parallelizes for every web page; in this case study, *Servo* uses 4 threads for all web pages. Even with a limited number of samples, both the regression and classification models beat *Servo* and bridge the gap between blind parallelism and optimal adaptive parallelism. Figure 1.1 depicts the styling times taken by *Servo* and *Our Model* against the *Optimal Model* (to which times are normalized) for the top 20 web pages in the Alexa Top 500 [3] list.

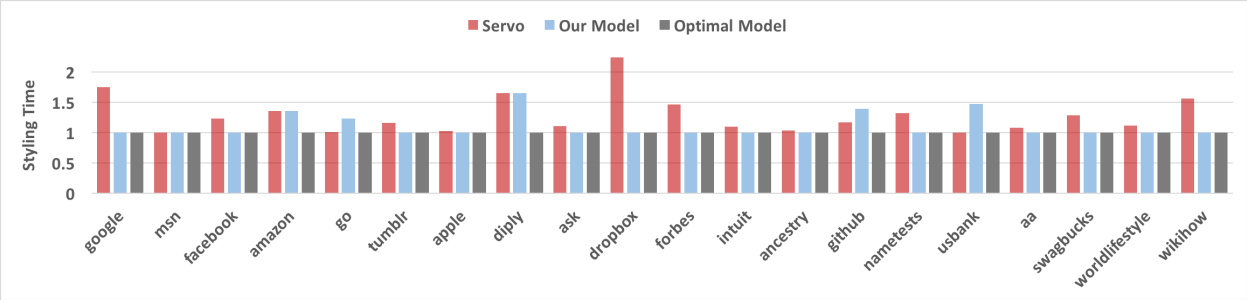


Figure 1.1: Normalized styling times of Servo, Our Model, and Optimal Model.

Concretely, we make the following contributions:

(1) **Workload characterization** – The workload of a browser is dependent on the web page. We study and analyze the Document Object Model (DOM) [7] tree characteristics of a web page and use them to characterize the parallel workload of the rendering engine (Section 3).

(2) **Performance-energy automated decision making** – Considering performance speedups and energy usage "greenups," [35] we propose algorithms that will decide the best thread configuration to use for a web page using one of three cost models. We can then use these algorithms to label our data set for classification-based models or to make decisions for regression-based models (Section 5).

(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 task for this web page? If so, (b) what is the degree of available parallelism? (Section 6) We evaluate both our classification and regression models (Section 7) against an *Optimal Model* and *Servo*.

Chapter 2

Parallelism in Servo

Servo [23] 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 [21], a new language designed by Mozilla Research specifically with Servo's requirements in mind. Rust provides a task-parallel infrastructure and a 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. In fact, Mozilla is in the process of shipping several of Servo's components to the Firefox browser through its Quantum project [17].

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

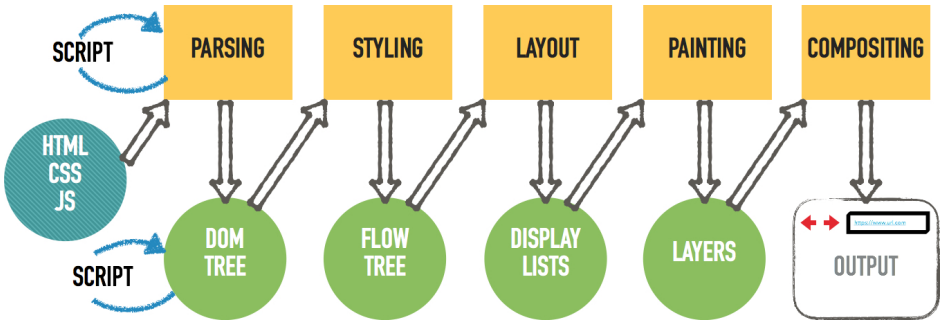


Figure 2.1: Processing stages and intermediate representations in a browser engine. The circles represent data structures while the squares represent tasks.

(W3C) and the Web Hypertext Application Technology Working Group (WHATWG). As such, the steps Servo uses in Figure 2.1 should be unsurprising to those familiar with the implementation of other modern browsers [9]. In this chapter, we briefly describe each of these phases and how Servo executes them. Appendix A.1 contains a detailed explanation of each processing stage.

2.1 Parsing HTML and CSS

In a new navigation to a resource, whether from a click on a link, entry in the address bar, or programmatic navigation via JavaScript, the first step in loading a site is retrieving and parsing the HTML and/or CSS files. The HTML document is translated into a DOM tree, the data structure used by browser's components to work with HTML elements. Each node of the DOM tree corresponds to an HTML element in the markup. The CSS file is loaded into style structures that will be used in *Styling*. JavaScript may execute twice, during parsing and after page-load during user-interactivity when it can modify the DOM tree. Servo uses html5ever [14], a high-performance browser-grade HTML5 parser written in Rust as a part of the Servo project.

2.2 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 is the subject of analysis in this thesis. Servo executes styling using parallel tree traversals, an approach similar to the one employed by Meyerovich et al. [40]. 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 nodes, requiring communication between parallel threads. The second half of this step is the construction of the flow tree.

2.3 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.

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

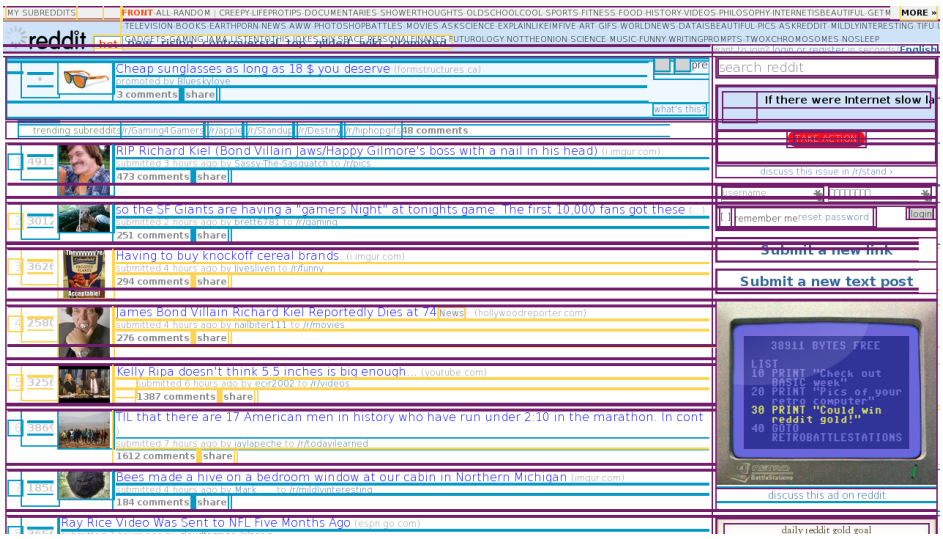


Figure 2.2: Parallel layout on `reddit.com`. Different colors indicate that layout was performed by a different thread.

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 2.2 shows one parallel execution with four cores rendering `reddit.com`.

2.4 Painting and Compositing

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 [28].

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. Figure 2.3 provides a visual example of how the Wikipedia page for the movie Guardians of the Galaxy* is partitioned and rendered.

*`https://en.wikipedia.org/wiki/Guardians_of_the_Galaxy_(film)`

Servo uses Webrender [30], a specialized GPU renderer designed for web content, for the purposes of painting.

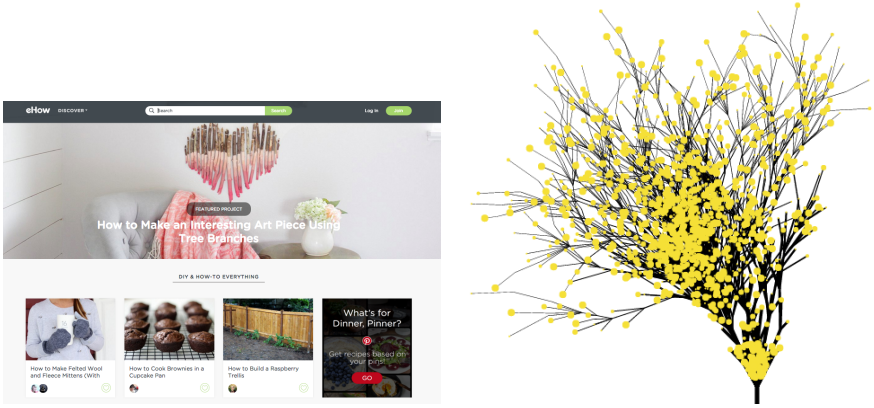
Figure 2.3: Parallel painting on a Wikipedia page using 3 threads. Different colors indicate that painting was performed by a different thread

In the rest of the thesis, we refer to *total* times of all the incremental passes performed in the styling stage. The sequential baseline performance of Servo's styling is nearly 2× faster than Firefox's (Table 1 of [31]). This speedup stems from the use of Rust instead of C++ as more optimization opportunities exist in Rust. Additionally, the parallel implementation of styling 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× 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 dynamically schedule the threads that are spawned (once) to perform the parallel tree traversals in styling and layout. The work-stealing scheduler is implemented by Rayon [18], a data-parallelism library for Rust.

Chapter 3

Parallel Workload Characterization

(a) `google.com`



(b) `ehow.com`

Figure 3.1: Contrasting types of web pages (left) and the visualization of their DOM trees (right).

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 3.1 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.

The right column of Figure 3.1 portrays a visual representation (created using Treeify [25]) 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 3.1 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.

Browser optimizations are primarily applied to the style application stage since it is the most CPU-intensive of all stages [9]. 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 [40]. Hence, I 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. I intuitively choose the following set of eight 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**)

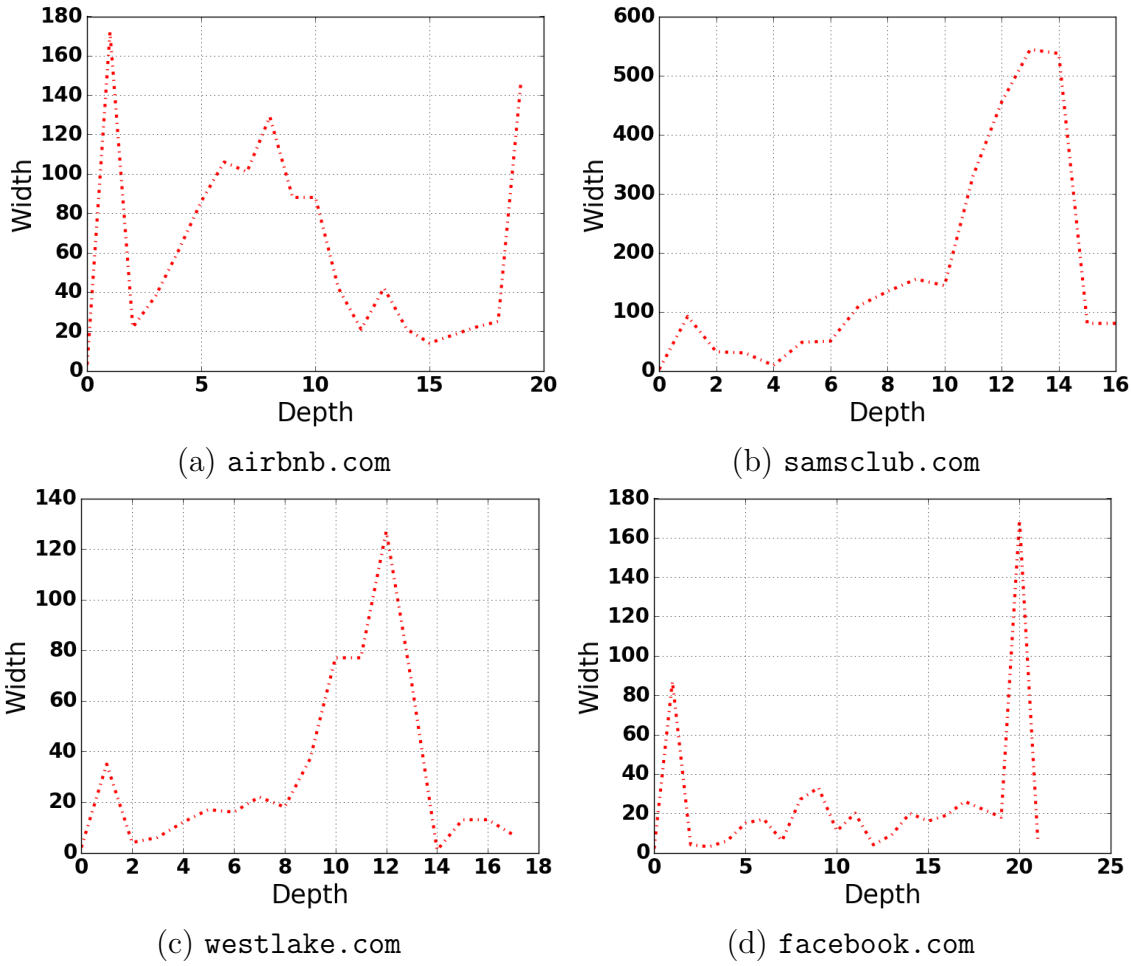
7. Maximum number of nodes at a level of the tree (**max-tree-width**)

8. Ratio of max-tree-width to average-tree-width (**max-mean-width-ratio**)

My intuition is that large* and wide trees would experience 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 (Section 2), I expect faster total traversal completion times on trees with a large number of leaves (captured by **number-of-leaves**) because the leaves enable parallelism. Even amongst wide trees, those that don't have abrupt changes in tree-width, or are less deep, should demonstrate faster parallel traversals (captured by **tree-depth**, **max-mean-width-ratio**) since the average workload would not be dominated by only one level of the tree and a smaller depth would mean lesser levels with low work. **DOM-size** captures the total amount of work while **avg-tree-width** captures the average parallel work on the web page. **attribute-count**, and **web-page-size** capture the general HTML information about a web page which corresponds to the workload contained in a web page.

To quantitatively analyze DOM trees, I plot the width of the trees at each of their depth levels. In Figure 3.2, I do so for `airbnb.com`, `samsclub.com`, `westlake.com`, and `facebook.com`. Using these figures, I relate my 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

*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.



(a) `airbnb.com` (b) `samsclub.com`

(c) `westlake.com` (d) `facebook.com`

Figure 3.2: Width Vs. Depth graphs of the DOM trees of different web pages (note that the scales of the axes are different).

favors parallelism. The performance experiments show that `samsclub.com` achieves 1.48× speedup with 2 threads and 2.12× speedup with 4 threads. `airbnb.com` achieves speedups of 1.2× and 1.43× 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× and 0.74× with 2 and 4 threads respectively. Similarly, `facebook.com` demonstrates slowdowns of 0.86×and 0.81× with 2 and 4 threads respectively.

Chapter 4

Experimental Setup

The World Wide 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, I use Google's Web Page Replay [29] (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. I 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. The testing platform includes a quad-core Intel i7-4980HQ running OS X 10.12 with 16GB of RAM on which Servo runs. Additionally, an Ubuntu machine runs the WPR framework. The Mac

is connected to the Internet through the Ubuntu's WiFi via a wired ethernet connection between the two systems; the Mac's Wifi is disabled. The WPR servers are listening and serving HTTP and DNS requests on the Ubuntu's ethernet port. By offloading the processing of network requests to an Ubuntu machine, I minimize the resource contention that would've been caused had the WPR servers been running on the Mac as well.

I collect the sample dataset from Alexa Top 500 [3] web pages in the United States during January 2016 and from the 2012 Fortune 1000 list [1]. I initially started with 1500 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, I have a working set of 535 web pages.

For the purposes of evalutions, I collect popularity data of the web pages. To do so, I use Alexa.com's Site Comparisons Tool [2] and record the Monthly Estimated Unique Visitors metric for each page in the dataset. Unique Visitors corresponds to the number of people that visited a site during the 30-day period; the same person visiting the website multiple times is only counted once [26]. The popularity distribution of the web pages follows a negative exponential model as we can see in Figure 4.1. I was able to fit the following model to the distribution

$$q(i) = e^{a-bi} \tag{4.1}$$

where $a = 17.95$ and $b = 0.02$. Some of the web pages in the data set didn't have estimates for Monthly Unique Visitors because Alexa didn't have enough traffic-data to calculate the metrics for those websites. So, I assign my own estimates for these websites based on negative exponential distribution, Equation 4.1.

For performance testing, I use Servo's internal profiling tool that spits out a CSV file con-

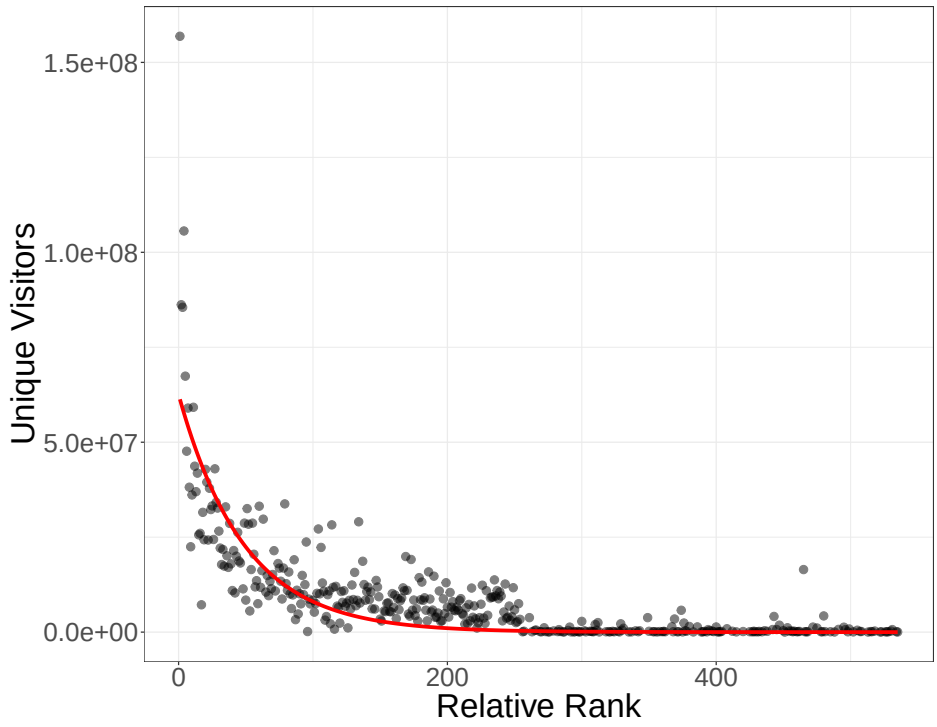


Figure 4.1: Popularity of a web page VS its relative rank in the dataset. The red line is the negative exponential curve that estimates the data.

taining user times of the *Styling* stage. For energy testing, I use Apple's powermetrics [16] to capture processor power usage. In both the energy and timing experiments, Servo opens a web page and terminates 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, I 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 its styling stage. Given that our platform is a quad-core, I used 1, 2, and 4 as the number of threads for each web page. However, Servo demonstrated non-repeatable behavior since it is still under

development. To account for repeatability, I 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 [38].

Chapter 5

Automated Decision Making

In this chapter, I propose tunable, automated decision-making algorithms that can be used with any web browser on any testing platform to classify web pages into different categories. These algorithms can be used for automated labeling when the goal is to train supervised classification models, thereby eliminating the need for a domain expert to manually and accurately label data. Or, these algorithms can be used for decision-making with regression models to choose the right thread configuration after using each regressor's prediction as input into these decision-making algorithms. The regression models are trained to predict the performance and energy usage of each thread configuration. For decision-making, I consider three cost models:

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

Although the styling stage is the focus of the analysis, I collected user times for the layout stage as well. An interesting observation is that an individual serial pass of the layout stage ranges between 1 and 55 ms. This range is much smaller than the 1 to 320 ms range of an individual serial pass of the styling stage. Hence, parallelizing tree traversals for the layout stage will most likely result in poorer performance due to thread communication and scheduling overheads. The results validate this analysis. On average, the time taken by an individual parallel pass for layout is 3.92 ms, 7.01 ms, and 9.82 ms with 1, 2, and 4 threads respectively. I also observe an increase in the average total times for layout with parallelization: 221.39 ms, 242.56 ms, 263.73 ms with 1, 2 and 4 threads respectively.

I collect the energy usage values of Servo in its styling stage using the available thread configuratoins. Although the data corresponds to the processor energy usage between the beginning and termination of Servo, these values are mainly affected by parallelization of styling because this stage constitutes the majority of the browser's execution time. I cannot obtain energy measurements at the granularity of function calls using Powermetrics [16], or any external energy profiling tool.

5.1 Performance Cost Model

In the Performance Cost Model, I consider only parallel runtimes to decide on the best thread configuration. Consider an arbitrary number of thread configurations where each configuration uses t threads. The values of t are distinct. I 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 decision-making process for a single web page:

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

Hence, if there are n thread-configurations, n possible decisions exist. If the decision on a web page is t, it means that using t threads achieves the best performance for that web page. Note that these decisions 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 decision-making on web pages using the Performance Cost Model.

Algorithm 1 Decision-making using the Performance Cost Model

```
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
        decision ← t
    else
        decision ← t_serial
    return decision
```

For the experimental testbed with 4 cores, three thread configurations are available: 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, I consider a 10% speedup to be significant. I attribute speedups less than 10% to noise. Hence, I 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.

5.2 Energy Cost Model

In the Energy Cost Model, I consider only the energy usage values to decide on the best thread configuration for a web page. The algorithm for making the decision is the same as Algorithm 1. Instead of using speedup values, I consider greenup [35] (energy usage improvement) values. Let y_t represent the energy consumed by t threads. For each thread configuration, I 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 the experimental platform is a laptop, I did not use the Energy Cost Model to classify our web pages. I will consider this model in the future for energy-critical mobile devices.

5.3 Performance and Energy Cost Model

In the Performance and Energy Cost Model, I consider both timing and energy usage values to decide on the best thread configuration for a web page. 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, I consider performance improvements through

parallelization to be useful only if the corresponding energy usage is lesser than an assigned upper limit. I label web pages using this cost model with a bucketing strategy as described below.

Similar to the classification in the previous two cost models, I 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, I 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 $p_{\min}$.
- 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 decision-making, I perform the following steps for each web page:

1. Ignore all the values of p_t that are lower than $p_{\min}$ and define PET buckets based on design considerations.
2. If the filtering results in an empty set, decide on t_{serial}. Otherwise, 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) Sort the PETs in the descending order w.r.t. p_t values.

(b) 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, 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, look at a lower bucket (one with the next highest speedups) and repeat the process. Do so until a PET satisfies the check against the energy usage limit.

4. If none of the PETs satisfy the condition, decide on t_{serial}. Otherwise, decide on thread configuration, t that corresponds 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 5.2 portrays a visual representation of the same.

Algorithm 2 Decision-making using the Performance and Energy Cost Model

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 $P_jP_{j+1}, \forall\, P_jP_{j+1} \in P \,\}$
$p_{\min}$: minimum threshold for significant speedup
procedure PERFORMANCE-ENERGY-LABELING
    $decision \leftarrow t_{\text{serial}}$
    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 all $PET_t \in PET''$ **do**
            if $e_t > E_j$ **then**
                $decision \leftarrow t$
                break
    return *decision*

For the case study with Servo, three thread configurations are 1 thread ($t = 1$), 2 threads ($t = 2$), and 4 threads ($t = 4$). I set the value of $p_{\min} = 1.1$ (from Section 5.1). Figure 5.1 depicts the histogram of Servo's p_2 and p_4 values. The histogram shows that, out of the significant speedups ($\sim$40%), the half point lies roughly at 1.3. Thus, I used two performance buckets: P_1P_2 and P_2P_3 where $P_1 = p_{\min}$, $P_2 = 1.3$, and $P_3 = 12.87$ (the largest observed speedup). For the first bucket, I set the energy usage increase tolerance, $E_1 = 0.9^*$ since a 10% energy usage increase can make a noticeable difference in overall battery life of a laptop. For the second bucket, I 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.

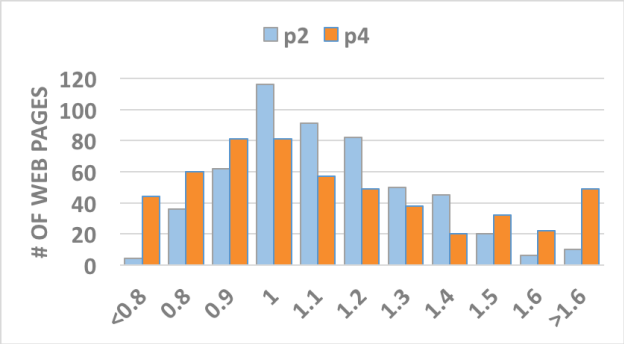


Figure 5.1: Histogram of speedup values. The bin labels are upper limits.

*These values can be tweaked based on design and device considerations. I chose these values for Servo on a quad-core Intel Ivy Bridge.

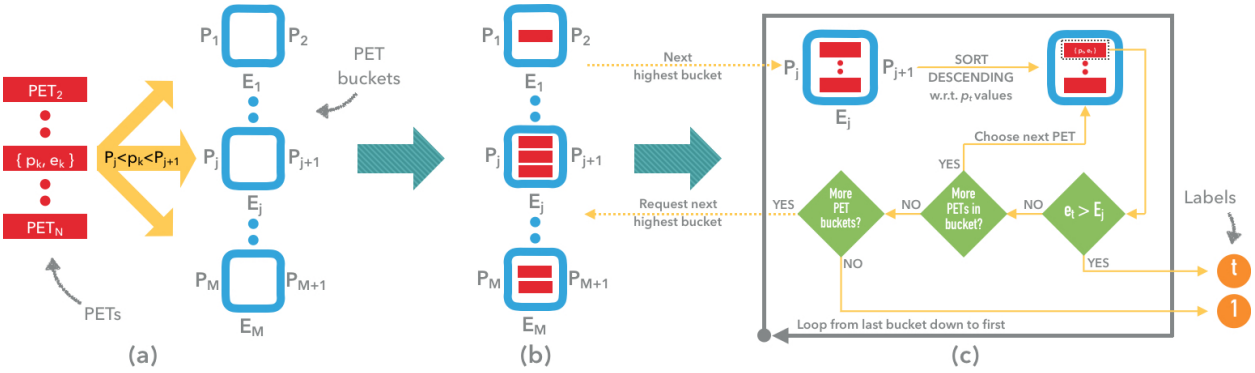


Figure 5.2: A visual representation of the Performance and Energy Cost Model's labeling algorithm for a given web page. (a) N PETs and M PET buckets. (b) Based on P_j and P_{j+1} values, the PETs are assigned to the right buckets. (c) Algorithm-flow to decide on the right thread configuration.

Chapter 6

Predictive Models

The 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 styling 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 amounts. Hence, the goal is to build a predictive model that allows a browser to decide the best thread configuration to use for its styling stage for any given web page by only looking at the web page's essential characteristics.

In this chapter, I treat the problem-at-hand as an offline, supervised learning problem. I observe linear relationships between the majority of the predictive features and the performance and energy output variables of each thread configuration. Additionally, after labeling the data using algorithms defined in Chapter 5 and after applying certain feature transformations, the classes were distinguishable in the pair-plots of the features. Hence, I choose to address this problem using regression classification approaches. For the purposes of eval-

uation, we use a 90-10% training-testing ratio *i.e.* 480 samples are used for training while 55 samples are used for testing.

6.1 Regression

In this approach, I train linear models that capture the relationship between the web page characteristics and performance and energy usage of the possible thread configurations. Once the linear models are constructed, we need to decide which thread configuration is best to use for a particular page. To do so, I use the Performance and Energy Cost Model's decision-making algorithm (Section 5.3) which uses the predictions of the linear regressors as its inputs. To train the regression models, I use a residual sum of squares as our loss function,

$$J_{ij}^{reg} = |R_{ij} - f(\theta_j, x_i)|^2 \tag{6.1}$$

where i is an index to the web page samples, j is an index to the regressors, R_{ij} is the true value of the function we are approximating, and f is the linear regressor. For each web page, f spits out a weighted sum of the web page's features, x_i where the weights are parameters, θ_j.

The goal is to find the best estimate of parameters, $\hat{\theta}$ such that

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \sum_{i=1}^{m} \sum_{j=1}^{k} J_{ij}^{regg} \tag{6.2}$$

where m is the total number of web pages in our data set, k is the number of output variables i.e. number of regressors.

In the data set, we have 3 thread configurations: $1, 2, 4$. For each thread configuration, we have a performance value and a energy usage value. Hence, $k = 6$ in the case study. To find

$\hat{\theta}$ (Equation 6.2), I train 6 linear regressors i.e. one each for x_1, x_2, x_4 (performance values, Section 5.1), y_1, y_2, and y_4 (energy values, Section 5.2).

After playing around with feature transformations, I observed approximately linear relationships between the features and the output variables in the logarithmic scale. To demonstrate, I portray some of the plots—Figure 6.1 shows the relation between the **dom-size** feature and the 6 output variables, Figure 6.2 shows the relationship between the various features and the performance output variable, x_4, and Figure 6.3 shows the same but against an energy output variable, y_4. Not all features observe linear relationships with the output variable. **tree-depth** and **max-mean-width-ratio** are such examples, as we can see in Figure 6.2. With $tree - depth$, the straight-lines on the graph shows that multiple web pages have equal depths but the variance in the parallel performance for pages with the same depth is high. Hence, $tree - depth$ alone is not a good descriptor of the parallel performance. Similarly, with $max - avg - tree - depth$ the parallel performance is better when the ratio is low *i.e.* the tree doesn't have abrupt changes in the tree widths. However,$max - avg - tree - depth$ alone is not a good descriptor of parallel performance and hence we don't observe a linear relationship.

Using the weighted loss function (Equation 6.1) we trained our models using R's linear model fitting function, `lm()` [19]. Figure 6.4 demonstrates the residual error values against the fitted (predicted) values. In the case of energy usage regressors, we observe that residuals get larger for higher values. The energy usage values observe a higher variance for larger values of the features (Figure 6.3) albeit linearly related with a majority of them. I attribute this larger variance to not being able to collect energy measurements at the granularity of functions; web pages whose features hold higher values could have larger processing times in other stages (I will study the relationship between web page features and the other stages of the browser in future work).

Once I have the predictions of these models for a set of features, I transform the predic-

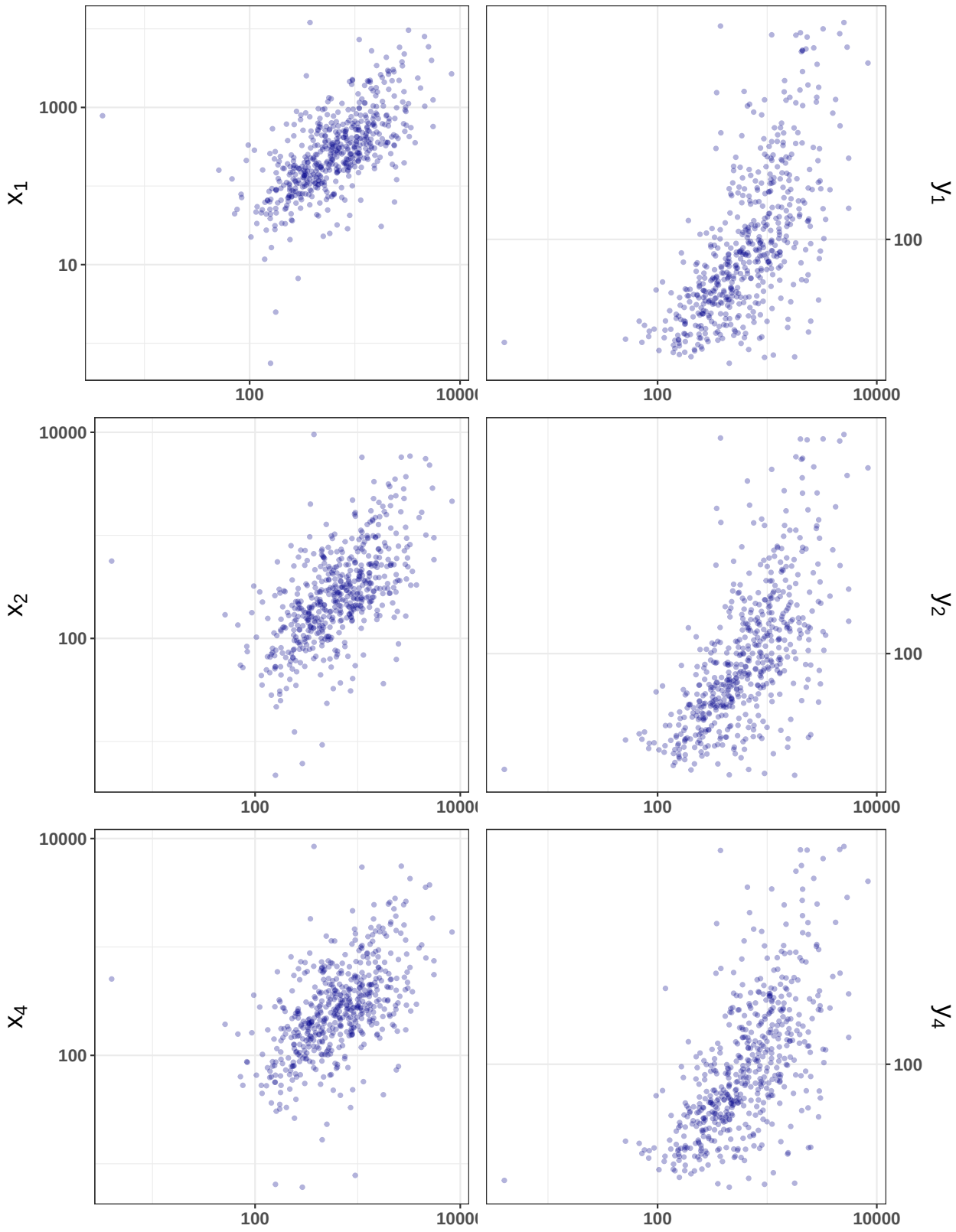


Figure 6.1: Relationships between dom-size and the 6 outputs in the logarithmic scale (base = 10).

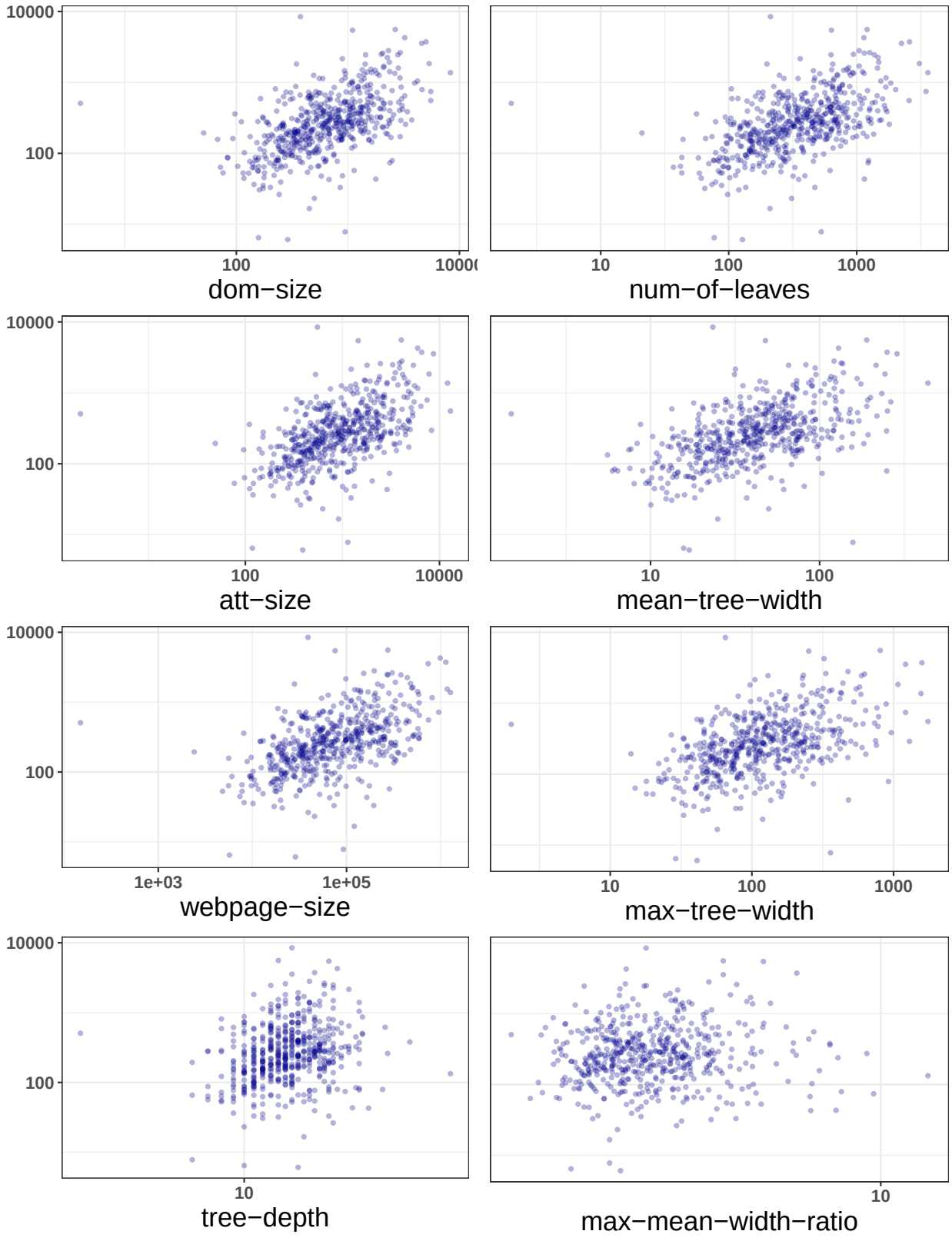


Figure 6.2: Relationships between all features and x_4 (performance with 4 threads) in the logarithmic scale (base = 10).

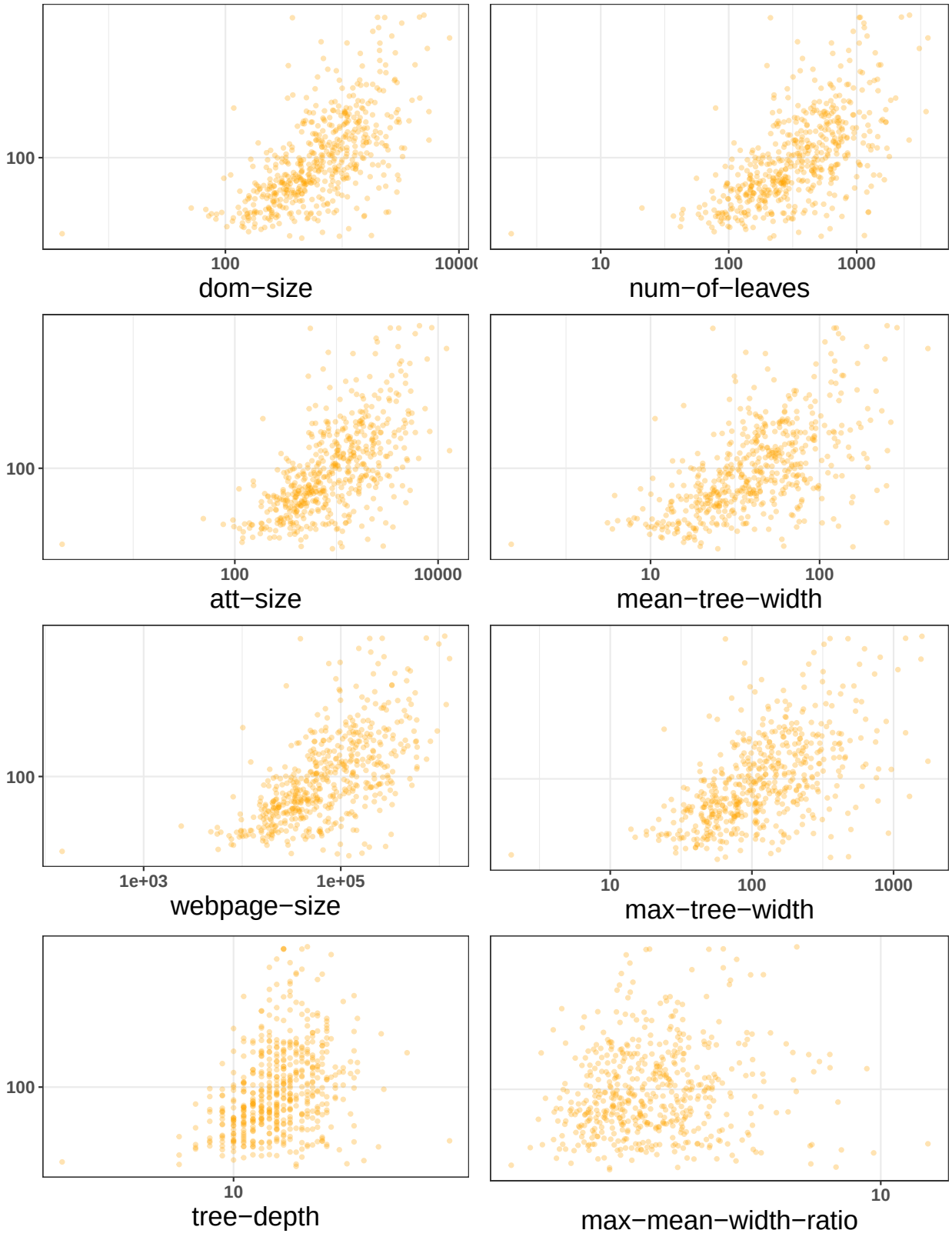


Figure 6.3: Relationships between all features and y_4 (energy usage with 4 threads) in the logarithmic scale (base = 10).

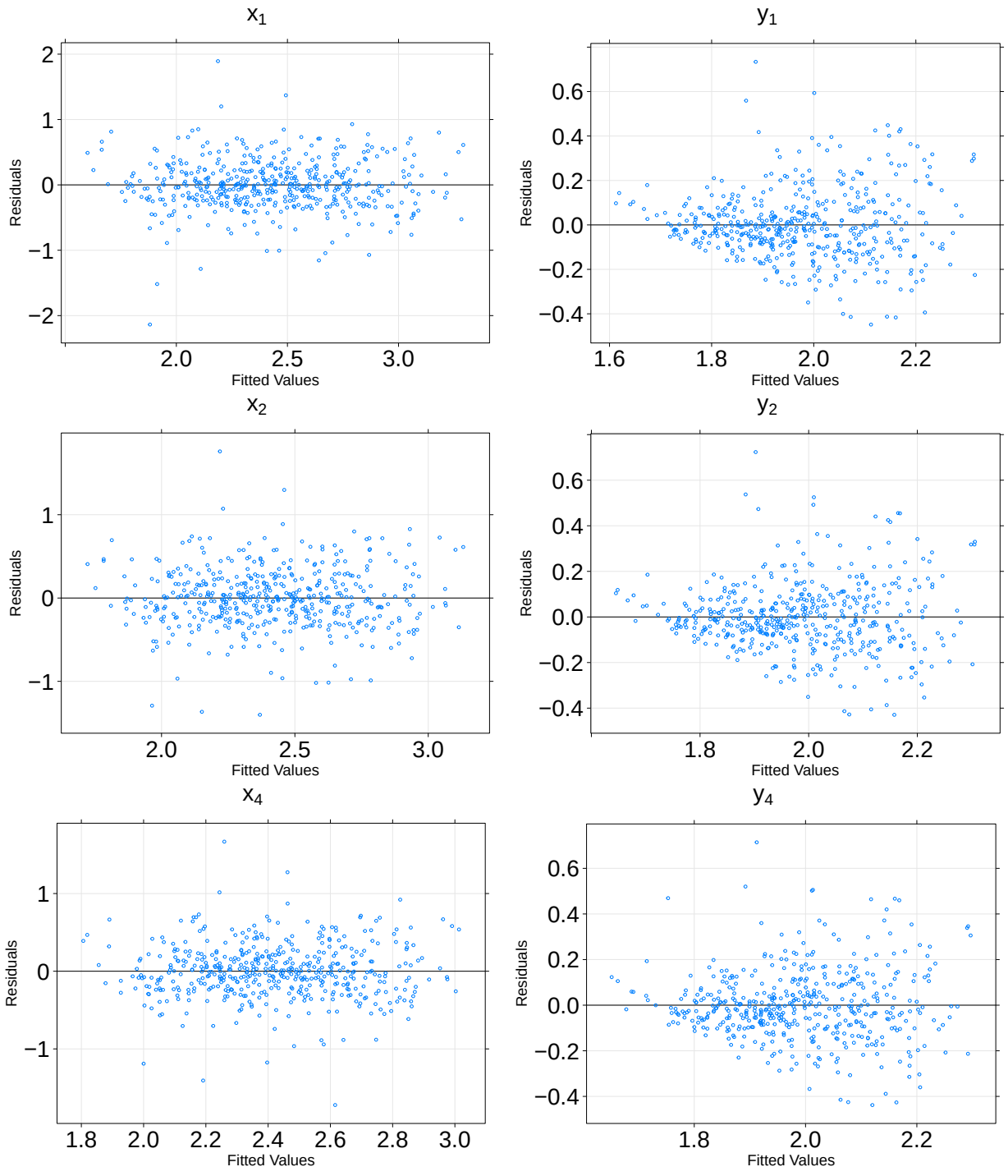


Figure 6.4: Residual Error VS Fitted Values for each of the 6 linear regressors

tions back into their original domain by exponentiating them and then feeding them into the Performance-Energy Cost Model's decision making algorithm. I evaluate this model's predictions on thread configurations in Section 7.

6.2 Classification

In this approach, I treat the thread configurations as classes to which each web page should belong. If thread configuration t is the best one to use for styling web page i, then that web page should belong to class t. Given the features of a web page, I use multinomial log-linear models to predict the probabilities that the given web page belongs to each of the possible thread configurations. The probabilities sum up to 1. Concretely, the loss function that we want to minimize is

$$J_{ij}^{class} = P_{ij}(\theta, x_i).R_{ij} \tag{6.3}$$

where i is an index to the web page samples, j is an index to the possible thread configurations, P is set of probabilities spit out by the log-linear model, and R can represent performance, energy usage or a combination of both depending on what the goal of optimization is. For instance, if the main priority is to optimize for performance, the R values can map directly to the performance values *i.e.* $R_{i1} = x_1$, $R_{i2} = x_2$, $R_{i3} = x_4$ because minimizing the sum would mean better performance. Similarly, if the main priority is to optimize for energy, the R values can map directly to the energy usage values where minimizing the sum would mean lower energy usage. When we want to optimize performance while considering energy usage, the value of R must represent a combination of the performance and energy usage values of the thread configuration such that the best thread configuration chosen by Performance and Energy Cost Model has the lowest value. When both performance and

energy need to be considered equally, I define R in the following way,

$$R_{ij} = (p_{ij} \,.\, e_{ij})^{-1} \tag{6.4}$$

where p_{ij} and e_{ij} are the speedup and greenup of the web page, i using thread configuration, j.

The log-linear model uses the first thread configuration, t_{serial} as the reference class and generates a linear model for the log-odds of every other thread configuration. In the case study, there are three thread configurations. So, we have two log-linear models: one for $t = 2$ and $t = 4$,

$$ln\left(\frac{P(2)}{P(t_{serial})}\right) = \theta_{00} + \theta_{01} \,.\, dom\_size + \theta_{02} \,.\, att\_size + \ldots \tag{6.5}$$

$$ln\left(\frac{P(4)}{P(t_{serial})}\right) = \theta_{10} + \theta_{11} \,.\, dom\_size + \theta_{12} \,.\, att\_size + \ldots \tag{6.6}$$

To train the parameters of the log-linear model, I get an initial estimate by regressing against the labels that the Performance and Energy Cost Model would choose for the samples in our data set. So, I first prepare the training dataset by labeling each web page using Algorithm 2. Once I have trained the model, I predict the best thread configuration to use by simply choosing the one with the highest probability. However, with such an approach we are not truly minimizing our loss function 6.3. Our goal is to find the best estimate of parameters, $\hat{\theta}$ such that

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \sum_{i=1}^{m} \sum_{j=1}^{k} J_{ij}^{class} \tag{6.7}$$

To implement a more robust classification model, I first visualized our dataset and applied feature transforms so that the different classes become more distinguishable than in their original feature-space. Figure 6.5 portrays a pair-plot of the features where the following transformation has been applied to the features: $log^3_{10}(X)$, where X corresponds to the set of features. I apply an additional logarithmic transformation to the max-mean-width-ratio feature. With the applied transformations, I observe distinguishable clusters of the distinct classes when they are function of two features. However, I also observe that a majority of the features are highly correlated and that the class-overlap is quite high, making this a hard problem for classification. A multinomial logit regressor that can learn linear decision boundaries is a good match to classify such data. Another classifier the could potentially be a good fit for this data set is the kNN classifier. However, in this chapter I focus on building a multinomial logit regressor and leave kNN for future work.

Using the weighted loss function (Equation 6.3) I trained the models using R's log-linear fitting function, `multinom()` [20] (available in the `nnet` package). `multinom` uses neural networks to implement its models. After obtaining an initial estimate of the parameters, I optimized the parameters to minimize the loss function defined in Equation 6.7 using the R's `optim()` function. `optim()` allows us to feed in gradients of the function that we are trying to optimize as well. I compute the gradients of J^{class} w.r.t. each of the 18 parameters and feed them into `optim()` as well.

I evaluate this model's predictions on thread configurations in Section 7.

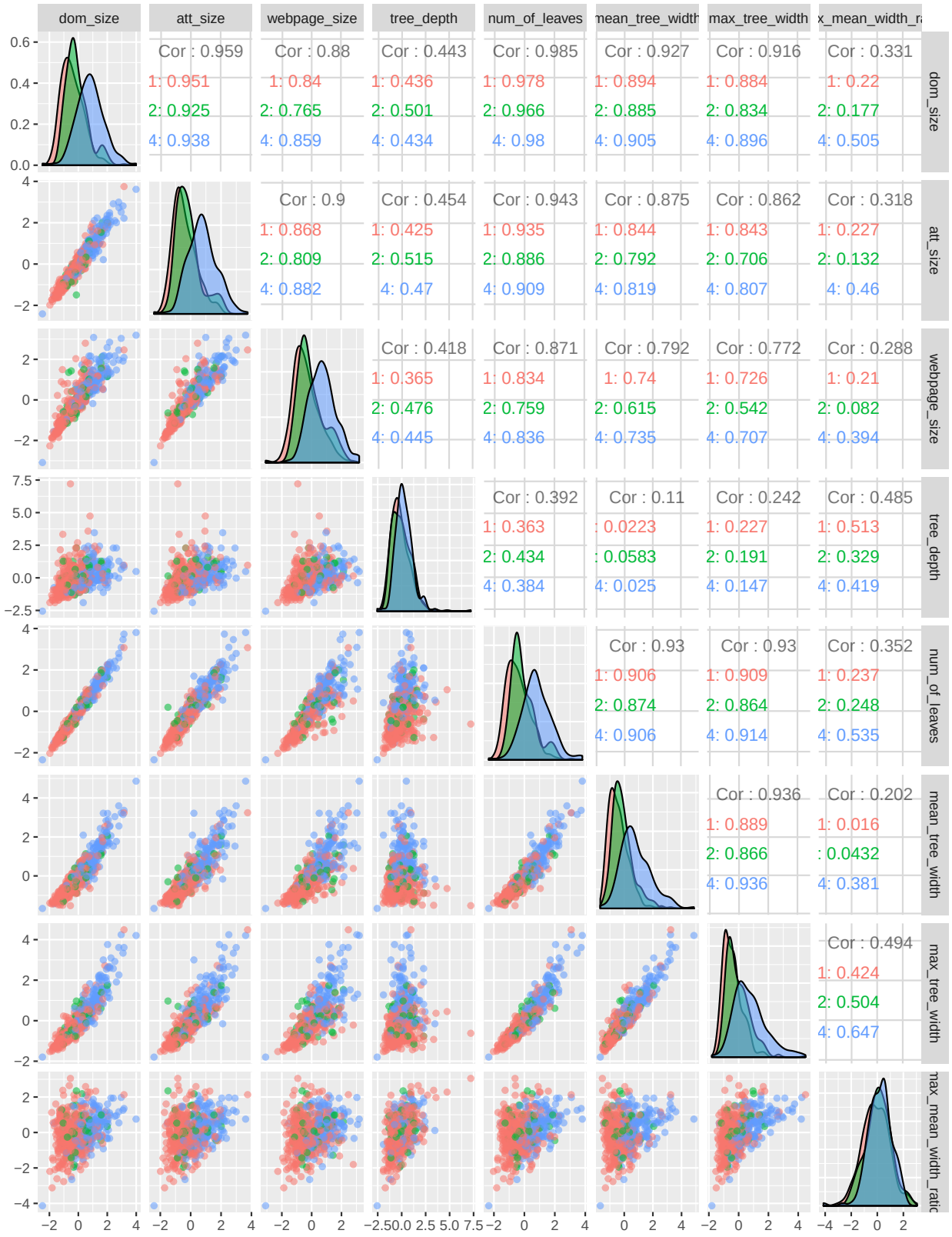


Figure 6.5: Pair-plot of features colored w.r.t. the class that the samples are labeled.

Chapter 7

Model Evaluation

In this chapter, I evaluate the efficacy of the models that we constructed in Section 6. The primary motivation behind creating such models is to save on performance and energy by choosing the best thread configuration to style a web page be it a serial execution or a parallel execution. Servo, as of now, blindly defaults to using 4 threads for styling. Hence, for each web page, I compare the performance and energy values corresponding to the thread configuration chosen by the model to that of the optimal thread configuration, t^*. More importantly, it is critical that the model is improving performance and energy usage on popular pages. Concretely, I compute the performance and energy savings of *Our Model* and *Servo* against an *Ideal Model*. *Our Model* corresponds to using the thread configuration chosen by the trained model for a sample, *Servo* corresponds to using 4 threads for each web page, and *Ideal Model* corresponds to the optimal thread configuration for the test samples. I define deviance from the optimal model for some metric, r as below:

$$Rd = \sum_{i=1}^{m_{test}} q_i \,.\, \frac{r^i_{t^*} - r^i_t}{r^i_{t^*}} \tag{7.1}$$

where q_i is the popularity of web page, i normalized over the test set. In other words, q_i represents the probability of the page being chosen from the set. t is the thread configuration chosen (by either *Servo* or *Our Model*). r can take on the value of either performance, energy or a combination of both. Equation 7.1 computes the expected percentage difference in a metric of *Servo* and *Our Model* from *Optimal Model*. Figure 7.1 portrays the results of such evaluation over a test set. Observe that both the classification and regression models beat *Servo* in reaching closer to the *Optimal Model*. The difference between *Servo* and *Our Model* is magnified in the Performance + Energy metric since the models are trained to minimize it in particular.

A more robust approach for evaluation, however, would be to compute the percentage difference in the expected performance and/or energy-usage instead of expected percentage differences because the latter is susceptible to outliers in the testing dataset and is hence more reflective of the samples it is being evaluated over rather than being reflective of the model itself. I re-define Rd from Equation 7.1 as

$$Rd = \frac{\sum_{i=1}^{m_{test}} q_i \,.\, r^i_{t^*} - \sum_{i=1}^{m_{test}} q_i \,.\, r^i_t}{\sum_{i=1}^{m_{test}} q_i \,.\, r^i_{t^*}} \tag{7.2}$$

Figure 7.2 shows the results of such evaluation over the same test set. The models perform significantly better than *Servo* when both Performance and Energy are need to be optimized together.

Both Figure 7.1 and Figure 7.2 show that *Our Models* are able to successfully predict the performance and energy improvements for the wide variety of pages and are able to bridge the gap between Servo's blind parallelism and the required adaptive parallelism. Additionally, our models achieve 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`).

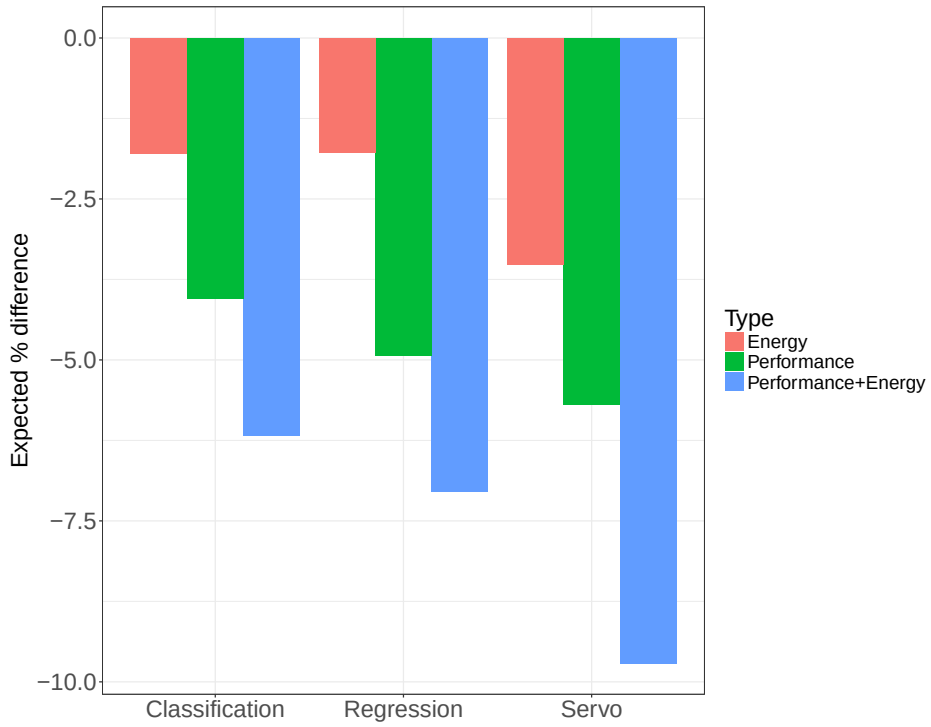


Figure 7.1: Expected percentage difference from the *Optimal Model* in the three metrics.

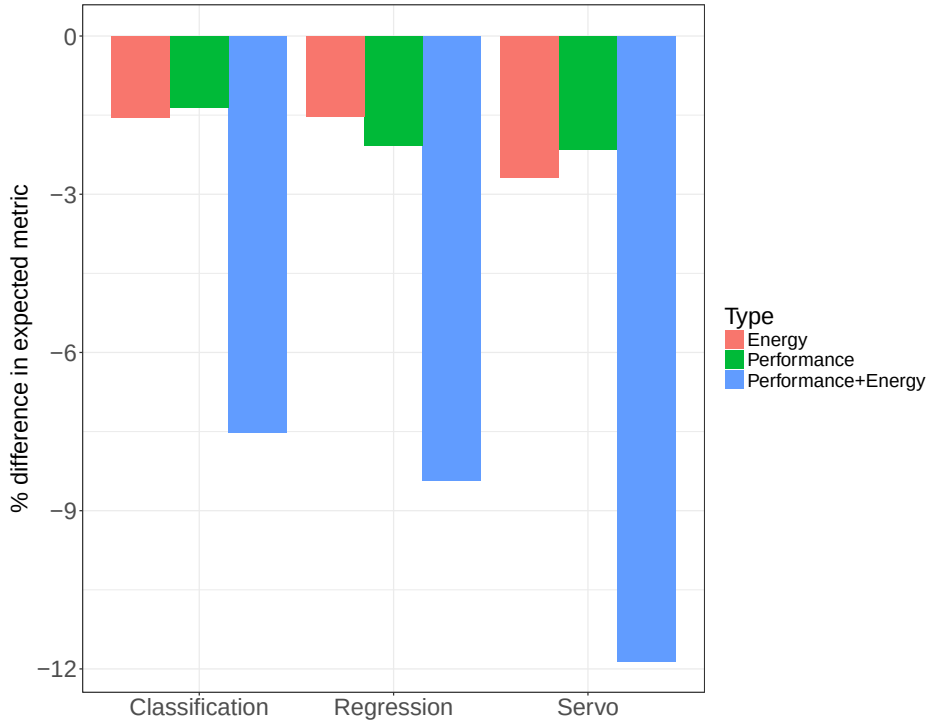


Figure 7.2: Percentage difference in the expected performance, energy-usage, and a combination of both from the *Optimal Model*.

Chapter 8

Survey on 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 prowess. The commodity browsers today such as Googles Chrome and Mozilla's Firefox exploit concurrency at the browser-component level by launching a process per tab. However, the computation within each task itself is not parallelized. 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. Multiple recent research projects exist both on the parallelization of browser-tasks and complete parallel browser implementations as well. To the best of my knowledge, Mozilla Researchs Servo is the only active parallel browser research project today. Below I outline existing research on parallelizing and analyzing browser-tasks.

8.1 Parallel Browser Implementations

In this section, I describe work pertaining to complete parallel browser implementations.

8.1.1 Gazelle

Title: *The Multi-Principal OS Construction of the Gazelle Web Browser* [43]

Venue: *USENIX Security Symposium 2009*

Motivation: Today's browsers allow the user to open up multiple web pages simultaneously in the form of different tabs or windows. Each of these tabs can run different web applications such as banking or social media utilities. It is the responsibility of the web browser to ensure that these applications are not able to access each other's data. Hence, the browser must maintain security across the various domains that are being accessed by the user at the same time. The browsers, such as Internet Explorer (IE) 8 and Chrome, with a multi-process architecture perform security-checks during the content-processing stages (such as parsing, styling, etc.). Wang et al. [43] were, however, able to break the security policies of these browsers in certain test cases.

Work: The authors present *Gazelle*, a research prototype browser, that performs its security-checks outside of any content-processing tasks. They do so using an approach similar to how an OS manages security between its multiple user accounts—not allowing one user to access the data in another user's account.

Figure 8.1 shows Gazelle's architecture. The **Same Origin Policy** (SOP) [22], states that two pages have the same origin if the protocol, port and host are the same for both pages. Gazelle spawns a process for each *principal* defined by SOP. The processes are isolated (sandboxed) so that they cannot interact with the underlying OS and must use system

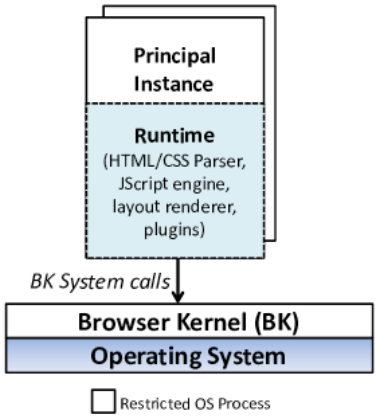


Figure 8.1: The Gazelle architecture [43]

calls provided by **Browser Kernel**. The Browser Kernel runs in a separate OS process and essentially functions as an OS, managing cross-principal protection on all resources, including network and display. This way, the security-checks are moved out of the content-processing tasks. Each Gazelle-process hosts its own processing components. This architecture is similar to that of Chrome but differs in isolation definition. If a web page contains resources from `ad.datacenter.com` and `user.datacenter.com`, Chrome would put them into the same process since it would consider them to be from the same site: `datacenter.com`. Gazelle, on the other hand, would put them into separate processes since they, according to SOP, have different origins. Such an architecture poses problems on cross-origin script source, display protection and resource allocation. In the paper, they detail their solutions for the first two problems and leave the resource allocation problem for future work.

Evaluation: The Gazelle browser was implemented using C# on Windows Vista with .NET framework 3.5 and evaluated on an Intel 2.66 GHz Core Duo with 2GB RAM and a 500GB SATA hard drive. The authors wrote about 5K lines of code to implement the Browser Kernel and used the web rendering components from IE 7. Their implementation was able to render 19 out of the top 20 Alexa sites (with some crashes and partial missing content) when compared against an unmodified IE 7 browser. Their evaluation showed high memory usage on news web sites, such as `nytimes.com`, due to the existence of multiple cross-origin

frames. Also, their implementation experienced a 471 ms and 3 s slowdown for `google.com` and `nytimes.com` respectively. They attribute these slowdowns to process-creation overhead and their unoptimized, instrumented prototype. They found that the median number of processes required to view a page is 4 (minimum of 1 and maximum of 28). They claim that Gazelle didn't demonstrate any difficulties when it created many (undefined) processes during normal browsing experience. Ultimately, the authors claim that it is realistic to turn an existing browser into a multi-principal OS that yields significantly stronger security and robustness with acceptable performance.

8.1.2 Adrenaline

Title: *A Case for Parallelizing Web Pages* [39]

Venue: *HotPar 2012*

Motivation: Amdhal's Law shows that parallelizing a single component of a browser is going to result only in minimal page load speedups. Although task-level parallelism can exploit thread safety along with concurrency, the overlap is bounded by the specifications of the Web and a web page's structure. Additionally, parallelizing individual components of a browser will benefit only those web pages and applications that make heavy use of those components. Since browser algorithms have been optimized over the years, and since their codebase is enormous, new parallel implementations will require a large amount of rewriting of existing commodity browsers.

Work: Mai et al. [39] propose that browser-developers should focus on *parallelizing web pages* instead of browser-tasks themselves. To evaluate their proposal, they design *Adrenaline*: a server-client prototype system to render multiple components of a web page in parallel. Adrenaline comprises of two computing systems: a server-side preprocessor and a client

browser (on some device). The work flow of the system is as follows:

(1) When the user requests for a web page on the browser, the client issues a request to the Adrenaline server.

(2) The server fetches the contents of the web page, optimizes and decomposes it into **mini pages** alongside a **main page**. Each mini page is a complete web page that consists of HTML, JavaScript, CSS, etc. that can be executed in an isolated process. The main page is responsible for assembling the mini pages. The server also computes a Bloom filter [33] for all the elements in each mini page. It sends the main page and the mini pages (along with their filters) to the browser.

(3) The browser then downloads, parses and renders each of the mini pages in separate processes running in parallel.

(4) The browser aggregates the content into a single display, synchronizing global data structures and propagating DOM and UI events to maintain correct web semantics. This is done by the main page process.

To facilitate normal user interaction, the main page captures external UI events and reroutes them to the right mini pages. This communication is done through IPC channels.

JavaScript is isolated into a single mini page: the main page. When JavaScript needs to access DOM states in the mini-pages, it requests and merges the DOM trees from the correct mini-page. The mini-page serializes its entire DOM tree and sends it back to the main page. The mini page is terminated. With the DOM attached, JavaScript can continue working on modifying the DOM states. This would be useful only on pages in which the JavaScript acts on a subset of the DOM tree. In such cases, the browser can process the DOM elements not accessed by JavaScript in separate mini-pages and not be blocked by JavaScript execution unlike a traditional browser.

Decomposition of web pages on the server reduces the total amount of work for tasks like parallel layout and rendering. For decomposition, the server renders the web page and extracts information such as element sizes, bounding boxes, visual locations of the element on the page. Using this information, the page is decomposed according to a heuristic algorithm. After the browser loads the web page, it provides feedback to the server about unanticipated DOM merges so that the server knows how to adjust the decomposition of the same page in the future.

Adrenaline implements multiple micro-optimizations. Its decomposition tries to prevent DOM merge operations and also tries to simplify them when they must occur. Since JavaScript and DOM-tree-processing occur simultaneously, it ensures that the correct DOM state is used for JavaScript execution.

Evaluation: The Adrenaline server is an HTTP proxy that decomposes the fetched web pages on the fly. The Adrenaline browser used the WebKit rendering engine and the V8 JavaScript engine. Using OProfile, they instrumented the browser to derive the time spent in the different components of the browser. Mini pages were implemented as browser plugins. The evaluation platform was a 400 MHz quad-core ARM Cortex-A9 with 768MB of RAM (CoreTile Express board). They compared against an unmodified QtBrowser (WebKit-based). They test on Alexa's top 170 web sites and observe that Adrenaline reduces the page load time by 1.75 s on average, 1.54× speedup on average. However, the authors failed to define what they considered to be page load time in their system. The authors don't report the fraction of time spent in the server which is critical since the server itself renders the web page to get information about the web page in order to decompose the web page for the browser. This, in my opinion, is an inherent overhead.

8.2 Parallelizing Browser Tasks

8.2.1 Parallelizing Firefox's Layout Engine

Title: *Towards Parallelizing the Layout Engine of Firefox* [32]

Venue: *HotPar 2010*

Motivation: Badea et al. [32] profiled the Firefox browser extensively and discovered that the Layout engine of Firefox accounts for 40% of its total execution time (on Intel platforms). CSS rule-matching was the hottest part of the layout, taking 32% of the execution time within the layout engine.

Work: The authors parallelize the CSS rule-matching algorithm but not the CSS selector-matching algorithm. They track speedups for the entire page-load process and not just the styling process alone. Their work deals only with descendant and sibling selectors (Section A.2.2). They decide to do so because, according to their profiling on the Zimbra Collaboration Suite and Firefox's page load benchmarks, 99% of the selectors are descendant selectors. For descendant selectors, all the ancestors of the DOM-tree-node-in-question need to be looked at to determine if the rule applies to the node i.e. the ancestors need to be checked if they match the parent-type specified in the rule. They also observed that in a vast majority of the cases, the end result of the iterations for a descendant selector lead is a non-match i.e. the rule doesn't apply to the DOM tree node in question. To exploit these observations, they implement a parallel version of rule-matching for descendant selectors.

For a (descendant selector) rule-matching that needs to be matched against a DOM tree node, the maximum number of iterations to execute is equal to the number of ancestors of the node. They divide the number of ancestors by the number of available threads and assign each thread a chunk of the ancestors to match the parent-type against. This involves

speculation since the authors assume that the thread that works on the earlier iterations doesn't match. The speculated work that proves to be unnecessary is discarded. The authors made sure that speculative work doesn't cause any side effects (no detail on how this is done). Each worker thread checks to see if another thread has already found a match to prevent extraneous computation (no detail on when the check occurs). They implemented their algorithm with three tunable parameters: number of worker threads, minimum number of rule-matching iterations for a thread, threshold on the workload size that needs to be met before enabling the parallel CS rule matching.

Evaluation: The authors use VTune [27] and VProf (value-profiling packing within Firefox's source codebase) with Firefox to measure performance and trace the rule-matching iterations. They evaluate on the Zimbra Collaboration Suite (ZCS) and Firefox page-load benchmarks using a quad-Core (2.93 GHz) Nehalem machine, with hyper-threading enabled and 2.49GB RAM. Their results show that 2 worker threads obtain the best results. On the Firefox page-load benchmarks, a maximum speedup of 1.84× was observed with an average of 1.1×. On the ZCS benchmark, 50% of the websites yield performance improvements with a maximum speedup of 1.8×. They consider these to be significant speedups since they are parallelizing only a certain part of the whole browser pipeline but are observing speedups on the entire page load process. The ZCS benchmark is more JavaScipt oriented that is why their algorithm showed lower benefits. This reiterates the point of the Adrenaline authors who state that parallelizing specific browser tasks will benefit only certain type of web pages and not all types. The authors claim that such parallelization will witness higher speedups as the complexity of the web pages continuously increases.

8.2.2 Fast and Parallel Webpage Layout

Title: *Fast and Parallel Webpage Layout* [41]

Venue: *WWW 2010*

The motivation of the work is the same as that in Section 8.2.1.

Work: The primary goal of this paper was to build a parallel web browser through the use of faster and parallel CSS selector-matching, layout and font-handling algorithms. Here, I will detail the algorithms only for the first task. The authors approach the layout solving problem by specifying their own styling language, which is impracticable to extend to render the real-world web-content. Font-handling is out of the scope of this paper.

CSS selector-matching: The authors innovations improve on parallelization and memory locality. The input to this algorithm are an object representation of the CSS style sheets and the DOM tree (of the whole document). The output is the Render tree. The style sheet consists of rules. The authors didn't implement any cascading (when multiple, conflicting rules match a DOM tree node) rules.

The authors define their own selector language: a rule with a selector s matches a document node n if the path from the document root to n matches the selector s. This corresponds to the descendant selectors in CSS. The authors claim that they found this common selector subset to comprise 99% of the rules encountered on Alexa's popular sites. This is consistent with the findings of Badea et al. [32]. Meyerovich et al. translate the selector subset that they deal with to a regular expression. For the selectors outside this regex definition, the unoptimized algorithm is used.

Their matching algorithm first creates hashtables associating HTML elements with selectors. Next, using 3 passes over the DOM, the nodes are matched against the selectors. Finally, a post-pass is performed to format the results. The authors use the following optimization techniques from the WebKit layout engine:

- **Hashtables** — They create 3 hashtables, one for the tag, class and id instances in the

document. In the selector `p img`, only the `img` HTML elements need to be checked against this selector. Iterating through the rules, the selectors are associated to right tag, class or id. This way, each node need not look at all the selectors in the style sheet, only the selectors that are associated with it in the hashtable.

- **Right-to-left matching** — For each rule in the stylesheet, the selector is processed from right to left i.e. in the DOM tree node, the child if first found and then the parents are iteratively checked to find if the rule matches. In `p img`, `img` is first processed.

The authors then use the following optimizations of their own:

- **Redundant selector elimination** — The authors claim that due certain weak abstractions in CSS, the stylesheets often contain multiple rules that use the same selectors. The authors aggregate rules with the same selectors during the hashtable creation process. This is a memory optimization.

- **Hash tiling** — Without this, the 3 hashtables would be accessed randomly generating cache misses since all the three tables and the DOM tree cannot all fit within the L1 and L2 cache. Hence, they make 3 passes and in each pass only one of tags, ids or classes are matched. This is a memory optimization.

- **Tokenization** — They replace the strings of HTML tags, classes and ids with unique integers to decrease size of data structures. This is a memory optimization.

- **Parallel document traversals with random load balancing** — The 3 passes on the DOM for each of tag, id and class are executed using parallel tree traversals. The authors map the DOM tree to an array of nodes (randomly) and use a work-stealing library to assign chunks of the array to each thread. The randomness is to load balance the work. The matching process can vary for different nodes and usually, the neighboring nodes have similar processing times.

Evaluating CSS selector-matching: The authors re-implement Safari's algorithm. Their re-implementation was within 30% of the original Safari implementation and hence considered their re-implementation to be representative of Safari. They compare their algorithm against that of Safari's on a 2.3GHz 4-core × 8-socket AMD Opteron 8356 (Barcelona). For each measurment, they performed 20 trials. Hash tiling alone achieves a speedup of 4×. Using Cilk++, they achieve an average speedup of 13× with 7 "hardware-contexts" (which I believe is equivalent to threads). Using Intel's TBB library, the average speedup is 55.2× using 6 hardware-contexts.

8.2.3 HPar: A Practical Parallel Parser for HTML

Title: *HPar: A Practical Parallel Parser for HTML–Taming HTML Complexities for Parallel Parsing* [44]

Venue: *ACM TACO 2013*

This is a very detailed journal paper. To keep the summary concise here, I will only talk about the ideas in the paper, some of the crucial implementation details and my takeaways from these ideas.

Motivation: Currently, parsing takes, on an average, 11.1% of the total page loading time (tested on Chrome on a MacBook Pro). However, when the rest of the browser-tasks are parallelized to obtain a speedup of 2x and 4x, sequential parsing would consume 20% and 30%, respectively, of the total page loading time. The goal of the authors is to remove the final bottleneck in the parallelization of the browser pipeline. Although simple techniques like pipelining and data-level parallelism can be employed to exploit parallelism benefits, the challenge is to design practicable techniques. Parsing HTML in itself is a difficult task—HTML is not defined by formal grammar even in HTML5 and it can contain scripts in other

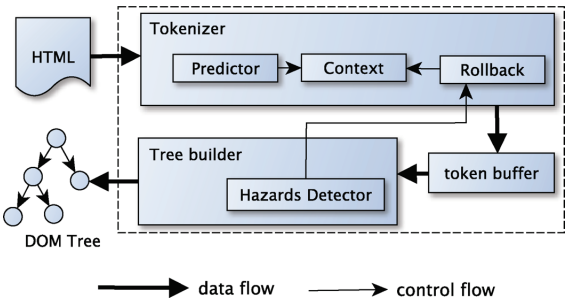


Figure 8.2: Speculative pipelined parser [44]

languages (inline JavaScript, CSS, etc.).

Work: Zhao et al. [44], design two parallel HTML parsers—one using speculative pipelining and another using speculative data-level parallelization.

According to the HTML5 specification, Tokenization (Figure A.3) cannot start processing new data elements before the Tree Construction finishes consuming the newly recognized token (the tree builder can alter the tokenization state depending upon its result). Additionally, the output of the Tree Construction stage is dependent on the tokenization state . Due to this cyclic dependency, direct pipelining won't work for the two-stage parsing process. Additionally, the processing of *self-closing* tags (such as `<br/>`), a unique case, requires cooperation between the Tokenization and the Tree Construction stages (detailed under Section 3.1 of the HPar paper).

Hence, the authors use **speculative pipelining** to break the dependencies between the two stages using the following approach:

(1) Predict the values of data items that cause dependencies.

(2) Process with predicted values.

(3) Verify prediction when the dependent data is produced. If it is wrong, trace back and reprocess the data from the correct state.

Figure 8.2 shows the architecture of this pipelined parser. They authors use an optimistic prediction approach i.e. no states change. They decided to do so through a statistical insight. The tokenization state is changed by the tree builder less than 0.01% of the times on Alexa's top 1000 web sites; among 117 HTML5 tags, 18.8% are self-closing. The **Hazard detector** checks for incorrectly predicted start states for both the Tokenization and the Tree Construction stages. The authors look into the HTML5 spec and identify all sites in the algorithm where a pipelining hazard may happen and incorporate all the cases into the Hazard detector. The Tokenization stage checks the *hazard flag* every time before it starts to produce the next token. If the hazard flag is set, the parser *rolls back* to where Tokenization just finished consuming the token that caused the hazard. This is done using a Snapshot data structure that contains data to recover. The *Token buffer* serves as the "pipe" in this pipelining scheme. The best performing buffer size was found to be one.

The authors also implement a **speculative data-level parallel** parser. They use a merging-oriented approach since a partitioning-oriented approach could result in load-imbalance. Additionally, a partitioning-oriented approach would have required a O(n) pre-parser step whereas the merge-step, although sequential, requires *O(log(n))* cost (the maximum-depth of the DOM tree with n tokens is approximately *log(n)*). At a high-level the document to be parsed is divided into multiple chunks and each chunk is parsed in parallel. The element of speculation occurs in this approach because only the first chunk knows the correct start-state. All the other parallel instances have to speculate their start-state. The authors decide on speculating the start-state to be the DATA since this was the case for 92.3% of the tokens in the top 1000 Alexa web sites. Whenever speculation is incorrect, the whole chunk is re-parsed. Additionally, they employ *smart-cutting* to ensure the that cuts in the HTML document are only before or after a tag instead of in between the characters of a tag, and *speculation-with-partial-context* to address cases where comments and `<script>` tags are broken by the cutting (Section 4.2 of the HPar delineates the implementations of these strategies). The crucial merge-step of their algorithm is quite intricate—the start/end tags of

one chunk need to be correctly matched with the end/start tags in the chunk that it is being merged with. The authors create a very simple language called LIST, that resembles the HTML structure using "start" and "end" tags to describe the structure of certain sections of a web page, and use it to describe the intricacies of their merge-steps (detailed in Section 4.1.3 of the HPar paper).

Evaluation: The authors implemented the two parallel parsers as modified versions of Jsoup [15], a well-maintained, open-source, stand-alone Java HTML5 parser. They demonstrate results for 10 real-world HTML pages on a MacBook Pro and a Nexus 7 tablet going from 1 to 10 threads against an unmodified Jsoup implementation. The speculative data-level parallel parser outperforms the speculative pipelined parser primarily due to the limited number of stages in pipelining. Their results show that pipelining parallelization is not a viable way to develop parallel parsers for HTML.

On the MacBook Pro, the data-level parallel parser achieves a maximum speedup of up to 2.4× with an average of 1.73×. On the Nexus, the same achieves a maximum speedup of 1.24× with an average of 1.13×. They attribute their sublinear speedup to the sequential tree merging step, which takes 5% of the overall parse time, to parallel overhead and to rollback overhead when speculation fails. Their speculation success-rate is 78% on an average.

Although the data-level parallel parser demonstrates appreciable speedups, in my opinion, it might be impracticable. In the real-world browsing experience, the whole HTML document is never available to the browser. In today's browsers the stages that follow parsing begin executing as soon as part of the HTML document is downloaded and parsed. The benefits of applying data-level parallelism on smaller, incremental amounts of HTML data is unknown and not explored in this paper.

8.3 Limit Studies

8.3.1 JavaScript Parallelism

Title: *A Limit Study of JavaScript Parallelism* [37]

Venue: *IISWC 2010*

Motivation: JavaScript everywhere on the Web and is the only component that adds the element of programmability to the web pages. 99.6% of sites online today use JavaScript. However, the dynamically-typed language doesn't allow compiler developers to easily optimize its execution. Its execution needs to be optimized in order to improve the user experience of web applications and also enable more computationally intensive tasks to run on web browsers. As mobile processor and screen technology progresses, the actual computation is constituting a larger and larger proportion of the total power usage for a mobile device. Fortuna et al. [37] address the slow computation issue by exploring the potential of parallelizing JavaScript applications.

Setup: The authors instrumented the interpreter in SquirrelFish and then used it with Safari to track the dynamic execution of events and JS functions while manually *interacting* with websites. Their benchmark suite comprised of full interactions with a subset of top 100 Alexa sites and 2 V8 benchmarks. Length of the logged interactions with the websites ranged from 5 to 20 minutes.

Work: The work in the paper is an offline analysis of parallel execution of JS events. The question of how to schedule events during runtime so that JS execution can be parallelized is however still open. This study is on register-based interpreters (see Section A.1.5). The register-slot can contain actual intermediate values (they call such elements as **virtual registers**) or references to object's hash table for lookups (they call them as **hash table**

lookups, even though they are in the register slots). After collecting the dynamic traces of JS calls, they analyze them offline and study what can be parallelized. They "parallelize" *events* individually by dividing them into **tasks**. A task is a consecutive stream of JS opcodes within an event. The delimiters used to divide a stream of opcodes into tasks are: a function call, a function return, an outermost loop entrance, an outermost loop-back-edge, or an outermost loop exit. The **critical path** is the longest sequence of memory or virtual register dependencies between tasks that occurs within an event. The length of this critical path defines the *length* of this event. All other data-dependent sequences can run in parallel to this critical sequence. They show an algorithm that computes the critical path for *each event* trace. During their analysis, they allowed events to run in the same order they were traced during the dynamic execution. The average task size in most benchmarks was around 17 opcodes or 650 cycles. They chose this small number since it was a limit study.

Speedup for a web page was calculated against the total execution time, T (in cycles) taken to execute the events during the interaction with the website. The length of the critical paths for all the events were then summed by which T was divided, returning the speedup. The authors assume that infinite processors are available.

Results: The authors calculate the mean potential speedup to be 8.91×. They observe that 52% of the loops in the test-suite iterated only 1 or 2 times. Hence, parallel-for loops wouldn't really benefit in most cases. Parallelizing functions themselves would prove to be more fruitful. The cases where parallel-for loops showed more benefits than parallel functions were usually the web applications that pertained more closely to scientific computing, such as FluidSIM [8]. This shows that parallel-for loops would benefit future web pages that are more compute intensive. Additionally, considering the small task size an insight is that certain tasks could be combined to amortize the parallelization overhead.

Chapter 9

Conclusion

The workload of a browser depends on the web page it is rendering and this thesis demonstrates that it can indeed be modeled—not by considering the platform that it is running on or by considering how the browser is designed but by simply looking at the web page's characteristics which are independent of the underlying implementations of the platforms. I use an offline, supervised learning approach to construct models that capture this relationship. Specifically, I characterize web pages using DOM tree and HTML characteristics that correlate to the styling task but are blind to the rendering engines implementation. I propose accurate and tunable, automated decision-making algorithms that categorize web pages into a user-defined number of thread configurations. These can be used for the purposes of labeling training data as well. Moreover, the algorithms account for tradeoffs between performance improvements and energy usage increases for multi-core processors. After visualizing relationships between the features and the output, I constructed two models: one is a simple linear function of the features for regression coupled with decision-making and the other is a multinomial log-linear model for classification. Both these models were trained on custom loss functions and beat *Servo* in optimizing for Performance and Energy achieving only 7.5% worse expected performance than the *Optimal Model*. Additionally, our models deliver per-

formance and energy savings up to 94.52% and 46.32% respectively when compared against *Servo*. This work is orthogonal to related work in that the 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. This thesis analyzes performance and energy trade-offs for a *parallel* browser while remaining agnostic to the browser implementation and execution platform, all within the complete execution cycle of a browser. To the best of my knowledge, this work is the first of its kind.

9.1 Future Work

The results of the work in this thesis primarily demonstrate the effectiveness and the necessity of modeling the relationship between web page features and performance and energy usage values. Although the modeling technique proposed is platform-independent, such a framework is still not practically feasible. If the browser-developers adopted such a framework, they would need to train the model for each type of platform that their browsers run on. Browsers today run on a wide variety of platforms: laptops, smart phones, smart TVs, smart watches, etc. Even within each type of platform, multiple types of processor chipsets exist. For example, the Intel chipsets have i3, i5 and i7 versions where each chipset has a different number of cores and hence a different number of thread configurations for the model to choose from. Even if the browser-developers had enough resources to train predictive models for their browser on each type of platform out there, the models cannot be relied upon by the browser-user. This is because the browser-user is almost always multi-tasking on their platforms and so the best thread-configuration that the model chooses may not be the best one given the current workload of the user, especially when the choice favors parallelism. Say, a browser-user enters the url that she wants to open up and switches to some other application that was already running. If there were 5 already running applications and

the model chose to use 4 threads for styling the page, the processor would encounter higher resource contention due to presence of more number of threads than cores and possibly cause the foreground application to respond slower since the browser in the background is styling the page.

Hence, a more practical approach would be to treat the problem-at-hand as an online learning problem with a pre-trained initial model. Online learning involves training a model on-the-fly with no prior knowledge (no training data). This means that the model will have to trade between "exploration" and "exploitation" in each iteration. Exploration involves trying the various actions that the model can take to acquire information about the expected payoffs of its possible predictions. Exploitation refers to using the information that the model has learned from its previous iterations to maximize its payoff.

In fact, the contextual multi-armed bandit problem maps well to the modeling task at hand. In this problem, an agent has to choose between arms that give rewards. To make a choice, the agent looks at a feature vector. Using the feature vector and the information that the user agent learned in the past iterations, the agent chooses an arm. The goal of the agent is to learn the relationships between the feature vectors and the rewards. In our case, the user agent is the model that is trying to capture the relationship between the web page's predictive features and the parallel performance and energy. The arms correspond to the thread configurations available. The feature vector contains the vector of the web page characteristics and some representation of the user's platform-workload. Each iteration refers to each new page being loaded.

Moving forward, we want to modify and map our loss functions of the classification and regression models to the contextual bandit problem. We intend to employ online learning on the models that were trained offline so that the online learning algorithms have a better initial state to start off from to reach the best set of parameters on the browser-user's platform. Such an approach enables the model to focus more on exploitation rather than exploration.

For the purposes of exploration during online learning, we can assign a certain probability that the model chooses a sub-optimal (from the model's perspective) thread configuration to either maintain its parameters or update them. With such a framework, the browser-developers will have to train models for a smaller variety of platforms such as Intel chipsets, AMD chipsets, Qualcomm chipsets, etc. and should successfully be able to port reliable models to their users because the model can adapt to the user's platform. Additionally, over time the browser will know how to predict rightly based on the workload of the user through its balance between exploration and exploitation.

Currently, we only model the styling task of the browser's tasks. Looking forward, we would also like to model the execution of the rest of the parallelized stages in the browser's pipeline to choose the best thread configuration to carry out its tasks. Additionally, the mobile space has more interesting thread configurations because of the presence of heterogeneous cores. Hence, we would like to first design and construct offline supervised models for a parallel browser and then ultimately employ online learning as well.

Bibliography

[1] 2012 Fortune 1000. http://booleanstrings.com/wp-content/uploads/2014/01/fortune1000-2012.xls.

[2] Alexa Site Comparisons tool. http://www.alexa.com/comparison.

[3] Alexa top 500. http://www.alexa.com/topsites/countries/US.

[4] BrowserHTML. https://github.com/browserhtml/browserhtml.

[5] CSS 2.2. https://www.w3.org/TR/CSS22/.

[6] Css snapshot 2015. https://www.w3.org/TR/CSS/#css.

[7] DOM definition. http://www.w3.org/DOM/#what.

[8] FluidSIM. http://www.fluidsim.de/fluidsim/index5_e.htm.

[9] How browsers work. http://www.html5rocks.com/en/tutorials/internals/howbrowserswork/.

[10] How browsers work. http://www.html5rocks.com/en/tutorials/internals/howbrowserswork/.

[11] How Fast Should A Website Load? http://www.hobo-web.co.uk/your-website-design-should-load-in-4-seconds/.

[12] Html5 parsing. http://w3c.github.io/html/syntax.html#syntax.

[13] Html5 validator. https://html5.validator.nu/.

[14] html5ever. https://github.com/servo/html5ever.

[15] Jsp. https://jsoup.org/.

[16] Powermetrics. https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man1/powermetrics.1.html.

[17] Quantum. https://wiki.mozilla.org/Quantum.

[18] Rayon. https://github.com/nikomatsakis/rayon.

[19] R's lm(). https://stat.ethz.ch/R-manual/R-devel/library/stats/html/lm.html.

[20] R's optim. https://cran.r-project.org/web/packages/nnet/nnet.pdf.

[21] The Rust language. http://www.rust-lang.org/.

[22] Same Origin Policy. http://www-archive.mozilla.org/projects/security/components/same-origin.html.

[23] The Servo web browser engine. https://github.com/servo/servo.

[24] Study: Load Times For 69% Of Responsive Design Mobile Sites Deemed Unacceptable. http://marketingland.com/study-load-time-69-mobile-sites-deemed-unacceptable-81126.

[25] Treeify. http://treeify.herokuapp.com/.

[26] Unique Visitors, Pageviews, and Visits. https://support.alexa.com/hc/en-us/articles/200462330-What-are-unique-visitors-pageviews-and-visits-.

[27] VTune performance analyzer. https://software.intel.com/en-us/intel-vtune-amplifier-xe.

[28] W3 painting order. http://www.w3.org/TR/CSS21/zindex.html#painting-order.

[29] Web Page Replay. https://github.com/chromium/web-page-replay.

[30] Webrender. https://github.com/servo/webrender.

[31] 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.

[32] 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.

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

[34] 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.

[35] 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.

[36] E. Fortuna, O. Anderson, L. Ceze, and S. Eggers. A limit study of javascript parallelism. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pages 1–10. IEEE, 2010.

[37] E. Fortuna, O. Anderson, L. Ceze, and S. Eggers. A limit study of javascript parallelism. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pages 1–10. IEEE, 2010.

[38] 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.

[39] 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.

[40] 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.

[41] L. A. Meyerovich and R. Bodik. Fast and parallel webpage layout. In *Proceedings of the 19th international conference on World wide web*, pages 711–720. ACM, 2010.

[42] 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.

[43] 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.

[44] 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.

Appendix A

Appendix Title

A.1 Browser internals

The **web browser** is a software application whose main goal is to render web-content that the user chooses to view. The location of the resource is specified using the **Uniform Resource Locator (URL)**; the browser requests the resource from the right server and displays it in the browser window. Web-content can be HTML, CSS, JavaScript, images, videos, PDF files, etc. A **web page** is essentially a **Hyper Text Markup Language (HTML)** document embedded with text and references to content. The HTML-elements of a web page can be "styled" (color, size, width, height, etc.) using corresponding **Cascading**

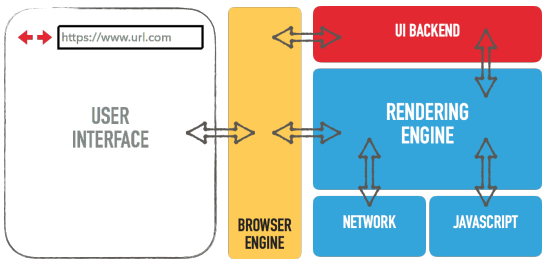


Figure A.1: Browser components

Style Sheets (CSS). The CSS can either be defined within the HTML document itself (inline defintions) or attached in separate `.css` files (better practice). **JavaScript** enables "programmability" on the web page, allowing complex user-interactions (apart from clicking links). JavaScript, similar to CSS, can either be defined in external `.js` files(better practice) or inline within the HTML. The way the browser should interpret HTML and CSS depends on the specifications specified by W3C. This way, web developers don't have to write different web pages for different web browsers.

Multiple popular web browsers exist today. Mozilla's Firefox, Google's Chrome, Apple's Safari, Microsoft's Edge and Opera are some of the notable ones. Although no standard specification exists for a browser's implementation, the high-level architecture of all browsers is essentially the same. Figure 2.1 depicts the general skeleton of any browser.

User Interface – This component encompasses everything in the browser window that is visible to the user *except* the area that displays web-content.

Browser Engine – This components mediates communication between the User Interface and the rest of the browser components. The User Interface needs to communicate with the UI Backend to render generic elements, such as buttons, that are dependent on the operating system.

UI Backend –This component is responsible for drawing basic widgets using OS-specific methods.

Network – This component handles and issues HTTP responses and requests to and from the browser.

JavaScript – This entails the JavaScript engine of the browser that either interprets or compiles native JavaScript code into machine code.

Rendering Engine – This is the defining component of a browser. The rendering engine

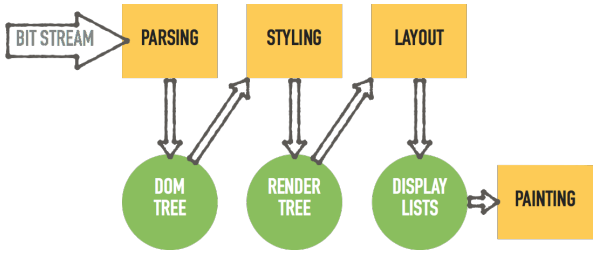


Figure A.2: Inside the Rendering Engine

is responsible for parsing the byte stream of any web-content and making it available to the end-user. This engine interacts directly with almost all other components of the browser except for the User Interface (it indirectly does so via the Browser Engine).

The general flow – When the user requests for a web page through the User Interface, the Network fetches the resource (usually in 8KB chunks). The HTML and CSS of the page are parsed and stored in data structures in the **Parsing** stage. The styles specified in the CSS are applied to the HTML in the **Styling** stage. During **Layout**, the absolute heights, widths and positions of the elements are calculated. Finally, the elements are displayed on the screen in the **Painting** stage. For any web page, these tasks can occur multiple times before the page is visible to the user i.e. Styling, for instance, does not wait for all of the document to be parsed. It will work incrementally on the chunks that have been parsed so far. All these stage occur in the Rendering Engine and are detailed in the following sub-sections. Figure A.2 shows the flow of tasks and data within the rendering engine. The yellow boxes represent tasks while the green circles represent the internal data structures used by the browser.

A.1.1 Parsing

Parsing means translating a document into a data structure that the browser software can use. The result of parsing is usually a tree of nodes. A browser's parser is responsible for parsing HTML and CSS content.

When parsing HTML alone, the parser might also need to handle parsing of CSS and JavaScript content because they can exist inline with the HTML through the use of the `<style>` and `<script>` tags, respectively. The HTML5 specification defines rules on HTML parsing and handling incorrect HTML [12]. However, it still doesn't use formal grammar. Figure A.3 shows the different components of HTML parsing specified by HTML5. The input to the parser is a byte stream of a HTML document and the output is a Document Object Model (DOM) [7] tree. The DOM is specified by the W3C as an interface that allows the different tasks to dynamically access and update the content, structure and style of documents.

The HTML5 parses comprises of two stages: Tokenization and Tree Construction. **Tokenization** corresponds to lexing i.e. the byte stream of the document is the input and the output is a stream of tokens (that belong to the HTML vocabulary). **Tree Construction** corresponds to the actual task of parsing i.e. the stream of tokens is the input and the output is the tree of nodes. Finite state machines are used to track the progress of Tokenization and the Tree Construction process. The states for the Tokenization stage are called the **tokenization state**, and the states for the Tree Construction are called the **insertion modes**.

As soon as the Tokenization stage detects a token, it sends the token to the Tree Construction stage. The Tree Construction stage organizes the parsing result of the token into the right location in the DOM tree (dependent on the tokenization state and the insertion mode). When the Tree Construction stage recognizes some scripts (*e.g.* JavaScript), the scripts are executed immediately. The result of the script could be some new HTML which would then be passed to the Tokenization. A lot of the parsing algorithm handles incorrect syntax. Unlike W3C's HTML5 Conformance Checker [13], the parsers don't notify the users about errors in their code. Instead, they try to fix them.

CSS [6] files are parsed into "style structures." While the grammar for CSS is defined in CSS

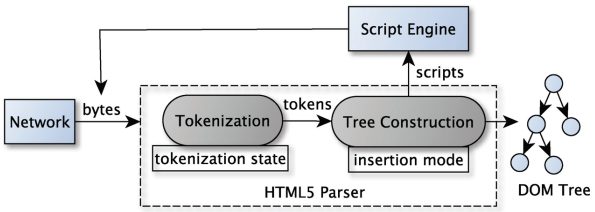


Figure A.3: HTML5 parsing model [44]

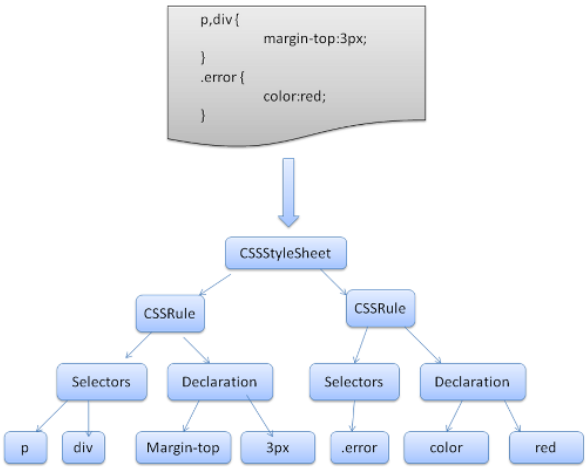


Figure A.4: CSS parsing example [10]

2.1, CSS 3 also specifies rules on tokenizing and parsing CSS while handling erroneous syntax. These structure consist of *CSSRule* objects. Each CSSRule object consists of *Selector* and *Declaration* objects (see Section A.2.2). Figure A.4 shows an example of CSS parsing.

The model of the web is **synchronous**. `<script>` elements will be parsed and executed as soon as the HTML parser thread comes across one. The parsing of the main document is halted until all of the content within the `<script>` tags are evaluated. If the source of the script is external, it is first fetched and then processed. Until the processing is complete, the rest of the document isn't parsed. With the `defer` attribute, the processing of any `<script>` elements can be execute only after the main document is parsed. HTML5 allows an additional `async` attribute that allows **asynchronous** execution of JavaScript (only external sources however) on a separate thread along with document parsing.

A.1.2 Styling

Styling is the process in which the rules specified in the CSS are matched to the appropriate nodes in the DOM tree. Once the browser has constructed the DOM tree, it must assign, for every element in the DOM tree, a value to every CSS property. This is the done according to the CSS specification (A.2.4). The output of this process is the **Render tree** (Figure A.2) which is essentially a DOM tree annotated with information about the style of each node. Each Render tree node represents a box on the screen (as per CSS specifications) and includes information like width, height and position. The relation between the Render tree and the DOM tree is, however, not one-to-one: non-visual elements such as `<head>` or those whose `display` attribute is set to `none` are not in the Render tree; elements with `visibility` set to `hidden` are in the Render tree; a single DOM tree node can correspond to multiple Render tree nodes–`<select>`, for instance, in the DOM tree corresponds to three Render tree nodes: one for the display area, one for the drop down list box and one for the button; when text is broken into multiple lines because the width is not sufficient for one line, new lines are added as extra Render tree nodes; some Render tree nodes that correspond to a DOM tree node are not in the same place in the tree. The root node of the Render tree is the "containing block" which is essentially the browser window display area. Hence the dimensions of the box at the root node are those of the browser window.

The Render tree needs to be constructed/modified whenever a part of the DOM tree is either created or updated. Each node of the DOM tree node needs to find out which styles apply to itself by looking at the rules (Section A) specified in the stylesheets. This is called **CSS selector-matching**. A naive way to do this would be to iterate through the whole list of CSS rules and check if a rule matches the node. For each rule, another series of iterations might need to be carried out to determine if the rule matches the node (Section A.2.2). This is called **CSS rule-matching**. The style structures (results of CSS parsing) are usually large constructs because of two reasons: they have multiple sources and today's web pages

use extensive amounts of CSS to enhance the user's experience with the web page. Hence, naive implementations of this stage can cause memory problems on top of slow performance due to copious amounts of selector-matching computation. Thus, browsers primarily focus on optimizing the matching task.

The word "cascading" in CSS refers to the order of preference in which style rules from multiple sources should be applied. The order is listed below in the ascending order of style-source priority:

(1) Browser declarations

(2) Browser user normal declarations

(3) Web page author normal declarations

(4) Web page author important declarations

(5) Browser user important declarations

The declarations within the same source are again sorted by **specificity** (defined in the CSS specification) to determine which styles override others in cases of conflicts.

A.1.3 Layout

Layout is the process of calculating the absolute position and size of the Render tree nodes on the display area. Before Layout, the Render tree nodes don't have any of this information. The coordinate system used to specify these positions is relative to the root of the Render tree. Coordinates of the top-left corner of an element are used to define location. Height and width are used to define the size of the element. The root is at (0, 0) and its dimensions are the visible part of the browser window. The following describes the flow of the Layout process:

(1) A parent node first determines its own absolute width (relative to the absolute width of its parent). This starts at the root where the absolute dimensions of the browser window are known.

(2) The children of the parent compute their heights. The (x, y) coordinates of these children are set by the parent.

(3) The height of the children are summed up. This sum equals the height of the parent along with any margins or paddings. The parent then propagates its height to its own parent.

During the Layout process, if a Render tree nodes needs to have a line break, the layout is stopped and the parent is asked to insert a new child. The parent then calls layout on its new child.

The Layout stage uses the *dirty-bit system*: for every addition/change to the Render tree, only the nodes that require layout are processed upon. Any Render tree node that has changed or been added marks itself and its children to be "dirty." There are two types of Layout:

- **Global Layout** – This occurs when Layout is triggered on the entire Render tree. This can be caused when a global style (*e.g.* font size of the browser) that affects all Render tree nodes changes or when the browser screen size is resized.
- **Incremental Layout** – This occurs when Layout is performed only for dirty nodes. This gets executed whenever a Render tree node tags itself as "dirty"— for instance, when new Render tree nodes are appended as a result of new data from the network.

All three steps of the Layout stage are executed only if Global Layout is occurring OR if the dirty bits of both the parent and the children are set.

A.1.4 Painting

In this stage, the Render tree is traversed and a **Display List** is constructed. A Display List is a set of commands for the compositing hardware to paint the screen. Painting can also be incremental when Render tree nodes are added/modified. The CSS specification defines the order of the painting process.

A.1.5 JavaScript

JavaScript (JS) is a dynamically typed, prototype-based, object oriented language. JS enables complex user interfaces on web pages. It can be both interpreted and compiled using Just-in-time (JIT) compilers. The current standard for JavaScript doesn't directly support concurrency.

JS stores its objects and properties in hash tables. Hence, getting, setting and adding a property to an object requires the interpreter or JIT compiler to either look up or modify that particular object's hash table. All JS engines, except V8 (used in Chrome), turn JS syntax into an intermediate representation of JS opcodes. V8 turns JS code directly into assembly code.

To manage intermediate data between the opcodes, commodity browser engines use two approaches: stacks and registers. **Stack-based** interpreters keep data in a stack, separate from the call stack. SpiderMonkey (used in Firefox) is stack-based. **Register-based** interpreters constitute an array of register slots (virtual registers) that are a level higher than actual machine registers. SquirrelFish Extreme (used in Safari) is register-based. Some research shows that using registers can outperform using stacks [37].

JS on the web is highly interactive and event-based. On the execution of an event, a single or a series of programmer-defined functions can execute. An event is either initiated by the

user or automatically by the user. After the functions have executed, the browser waits for the next event before executing more JS code (see Section A.3 for an example).

A.2 CSS

A.2.1 Specification

The CSS specification is defined by the W3C organization. The core of today's CSS specification is CSS 2.1. The newer specifications/modifications/corrections to CSS will be implemented in a modular format. The CSS 3 specification, for example, is the newest update to the CSS specification but, unlike in the past, its formal document contains only an update to a certain sub-section of CSS and not a complete re-write of the entire CSS specification. A second revision to CSS 2, CSS 2.2 is a work in progress [5]. A snapshot of the all definitions of the latest CSS (up until 2015) exists [6] (this includes CSS 3). Some features in CSS2 are not part of CSS 2.2, so not all CSS2 style sheets are valid CSS 2.2 style sheets.

A.2.2 Rule Structure

In a CSS stylesheet, multiple *style-rules* exist. Each CSS *rule* (*e.g.* h1 { color: red }) consists of two part: a *selector* (h1) and a *declaration* (color: red). The *selector* could also have a *declaration block*, which comprises of multiple *declarations* separated by semicolons. Each *declaration* has a *property name* (color) and a *property value* (red). Figure A.5 exemplifies the CSS rule structure. The *selector* can be an HTML tag, class or id. More often than not, conflicts between different property-values for the same HTML element exist; the CSS specification describes how to resolve the conflicts.

Multiple types of such *selectors* exist: type selectors, universal selectors, descendant selec-

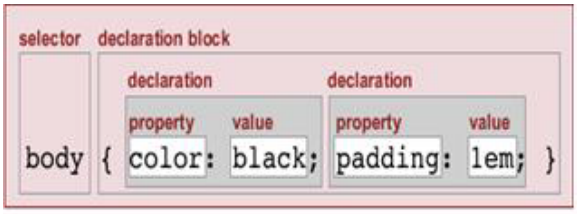


Figure A.5: CSS rule structure example [32]: the `<body>` elements in the web page will be colored black.

tors, sibling selectors, child selectors, class selectors, attribute selectors, etc. The selectors are essentially *patterns* comprising of simple selectors separated by *combinators*. The combinators are either white space, ">" or "+". The pattern defines a certain condition that an element needs to satisfy if the element is to inherit the style defined by that selector. If all conditions in the pattern are true for a certain element, the selector *matches* the element. This is called *pattern-matching*.

A *descendant-selector* is of the form A B—the rule described for this selector will match only when element-type B is a descendant of an element of type A. The rule-matching process for such selectors is carried bottom-up, starting from the child in the selector (to see if it is in fact a descendant of the parent described in the selector). Example: `ul em color: blue`. For the web document using this descendant selector CSS 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 documents tree) will be colored blue. This matching process is illustrated in Figure A.6, 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 [32].

A *sibling-selector* is of the form $A + B$—the rule described for this selector will match only if element-types A and B have the same parent in the document tree and element-type A

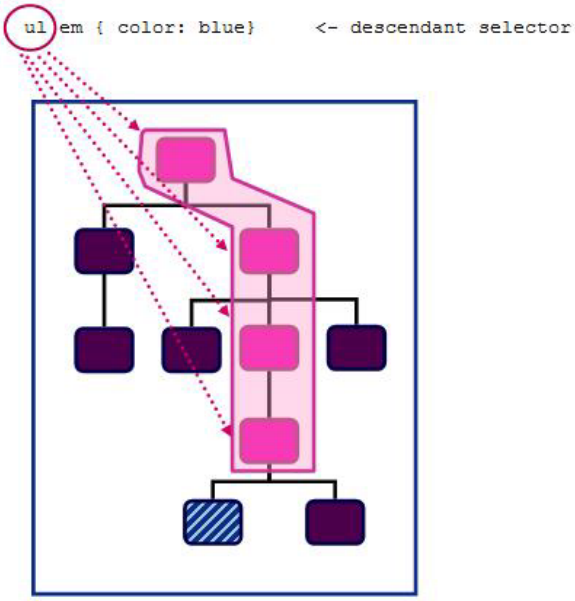


Figure A.6: CSS rule-matching example on the sample tree document [32]

immediately precedes element-type *B*.

A.2.3 CSS 2.2 Processing Model

In this model, a user agent (typically a browser) processes a source (typically an HTML document) by going through the following steps:

(1) Parse the source document and create a *document tree*. The document tree is a tree representation of the elements in the document. In the case of an HTML document, the document tree is the DOM tree.

(2) Identify the target *media type*. The media type reflects the *canvas* on which the document is going to be rendered. Typically it is a screen (in the case of a laptop) but it could also be "paged" (pages), "handheld" (for mobile devices), etc.

(3) Retrieve all style sheets associated with the document that are specified for the target media type.

(4) Annotate every element of the document tree by assigning a single value to every property that is applicable to the target media type. Properties are assigned values according to the cascading and inheritance mechanisms. Part of the calculation of values depends on the formatting algorithm appropriate for the target media type.

(5) From the annotated document tree, generate a *formatting structure* (*e.g.* the Render tree). Often, the formatting structure closely resembles the document tree, but it may also differ significantly, notably when authors make use of pseudo-elements and generated content. First, the formatting structure need not be "tree-shaped" at all – the nature of the structure depends on the implementation. Second, the formatting structure may contain more or less information than the document tree. For instance, if an element in the document tree has a value of 'none' for the 'display' property, that element will generate nothing in the formatting structure. A list element, on the other hand, may generate more information in the formatting structure: the list element's content and list style information (e.g., a bullet image).

The CSS user agent does not alter the document tree during this phase. In particular, content generated due to style sheets is not fed back to the document language processor (e.g., for re-parsing).

(6) Transfer the formatting structure to the target medium (e.g., print the results, display them on the screen, render them as speech, etc.).

A.2.4 Assigning CSS Property Values

Once the document tree is constructed, the user agent has to assign a value to every property for every element in the document tree. The final value of a property is the result of a four-step calculation: the value is determined through specification (the *specified value*), then resolved into a value that is used for inheritance (the *computed value*), then converted into

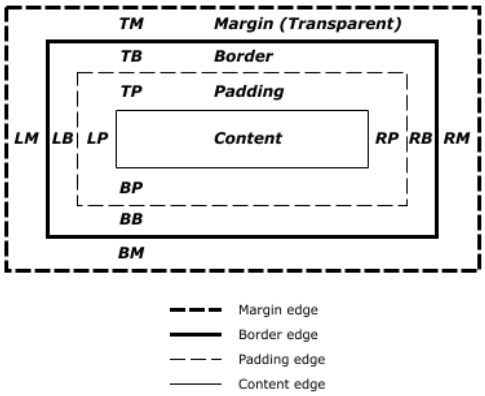


Figure A.7: The CSS 2.2 Box Model [5]

an absolute value if necessary (the *used value*), and finally transformed according to the limitations (such as integer-approximations of fractions) of the local environment (the *actual value*). These values are defined by the rules of *inheritance*, *cascading* and *importing*. The *specificity* of a selector is an important factor that needs to be computed for every selector in order to decide the right order cascading.

A.2.5 The CSS 2.2 Box Model

This box model describes the rectangular boxes that are generated for elements in the document tree and laid out according to the visual formatting model. Each box has a *content area*, such as text or image, and an optional surrounding *padding*, *border* and *margin* areas. Figure A.7 pictorially describes the box model.

The position of each box depends on the *positioning scheme*. The following positioning schemes are defined by setting the `position` property or the `float` attribute:

- Normal: the box is placed according to its position in the document tree.
- Float: the box is first placed using the normal positioning scheme but is then moved as far left or far right as possible.

- Absolute: the box is placed in an entirely different place than its place in the document tree.

Different types of boxes also exist. These can be *block* or *inline*. Blocks have their own box and are formatted vertically one after the other. Inlines are inside a containing a block and are formatted horizontally.

A.3 JavaScript Event-based Model

When a user clicks on a `<p>` HTML element, an `onClick` event is fired. If there is a function that is programmed to change the text of `<p>` on click, the function has to change the text of the `<p>` element and hence the DOM tree of the document. It does so by modifying the object's text through its hash table. Local variables needed to complete this text-modification action are stored in the virtual registers.