# Fast and Parallel Webpage Layout*

Leo A. Meyerovich †
lmeyerov@eecs.berkeley.edu
University of California, Berkeley

Rastislav Bodík
bodik@eecs.berkeley.edu
University of California, Berkeley

## ABSTRACT

The web browser is a CPU-intensive program. Especially on mobile devices, webpages load too slowly, expending significant time in processing a document's appearance. Due to power constraints, most hardware-driven speedups will come in the form of parallel architectures. This is also true of mobile devices such as phones and e-books. In this paper, we introduce new algorithms for CSS selector matching, layout solving, and font rendering, which represent key components for a fast layout engine. Evaluation on popular sites shows speedups as high as 80x. We also formulate the layout problem with attribute grammars, enabling us to not only parallelize our algorithm but prove that it computes in $O(log)$ time and without reflow.

## Categories and Subject Descriptors

H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*Benchmarking, graphical user interfaces (GUI), theory and methods*; I.3.2 [**Computer Graphics**]: Graphics Systems—*Distributed/network graphics*; I.3.1 [**Computer Graphics**]: Hardware Architecture—*Parallel processing*

## General Terms

Algorithms, Design, Languages, Performance, Standardization

## Keywords

attribute grammar, box model, CSS, font, HTML, layout, mobile, multicore, selector

## 1. INTRODUCTION

Web browsers should be at least a magnitude faster. Current browser performance is insufficient, so companies like Google manually optimize typical pages [14] and rewrite them in low-level platforms for mobile devices [1]. As we have previously observed, browsers are increasingly CPU-bound [8, 16]. Benchmarks of Internet Explorer [18] and Safari reveal 40-70% of the average processing time is spent on visual layout, which motivates our new components for layout. Crucial to exploiting coming hardware, our algorithms feature low cache usage and parallel evaluation.

Our primary motivation is to support the emerging and diverse class of mobile devices. Consider the 85,000+ applications specifically written for Apple's iPhone and iTouch devices [10]. Alarmingly, instead of just refactoring existing user interfaces for the smaller form factor, sites like `yelp.com` and `facebook.com` fully rewrite their clients with low-level languages: mobile devices suffer 1-2 magnitudes of sequential performance degradation due to power constraints, making high-level languages too costly. As we consider successively smaller computing classes, our performance concerns compound. These applications represent less than 1% of online content; by optimizing browsers, we can make high-level platforms like the web more viable for mobile devices.

Our secondary motivation for optimizing browsers is to speedup pages that already take only 1-2 seconds to load. A team at Google, when comparing the efficacy of showing 10 search results vs. ∼30, found that speed was a significant latent variable. A 0.5 second slowdown corresponded to a 20% decrease in traffic, hurting revenue [14]. Other teams have confirmed these findings throughout Facebook and Google. Improving client-side performance is now a time-consuming process: for example, Google sites sacrifice the structuring benefits of style sheets in order to improve performance. By optimizing browsers, we hope enable developers to instead focus more on application domain concerns.

Webpage processing is a significant bottleneck. Figure 1 compares load times for popular websites on a 2.4 GHz MacBook Pro to those on a 400MHz iPhone. We used the same wireless network for the tests: load time is still 9x slower

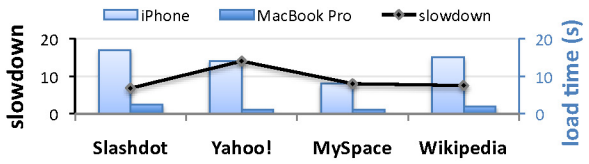



Figure 1: 400Mhz iPhone vs. 2.4Ghz MacBook Pro load times using the same wireless network.

*Research supported by Microsoft (Award #024263 ) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227).

†This material is based upon work supported under a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.


*WWW 2010*, April 26–30, 2010, Raleigh, North Carolina, USA.
ACM 978-1-60558-799-8/10/04.

on the handheld, suggesting the network is not entirely to blame. Consider the 6x clock frequency slowdown when switching from a MacBook Pro to an iPhone, as well as the overall simplification in architecture: the 9x slowdown in our first experiment is not surprising. Assuming network advances make mobile connections at least as fast as Wi-Fi, browsers will be increasingly CPU-bound.

To improve browser performance, we should exploit parallelism. The *power wall* – constraints involving price, heat, energy, transistor size, clock frequency, and power – is forcing hardware architects to apply increases in transistor counts towards improving parallel performance, not sequential performance. This includes mobile devices; dual core mobile devices are scheduled to be manufactured in 2010 and we expect mobile devices with up to 8 parallel hardware contexts in roughly 5 years. We are building a parallel web browser so that we can continue to rely upon the traditional hardware-driven optimization path.

Our contributions are for page layout tasks. We measured that at least 40% of the time in Safari is spent in these tasks and others report 70% of the time in Internet Explorer [5]. Our paper contributes algorithms for the following presentation tasks in CSS (Cascading Style Sheets [4, 12]):

**1. Selector Matching.** A rule language is used to associate style constraints with page elements, such as declaring that pictures nested within paragraphs have large margins. We present a new algorithm to determine, for every page element, the associated set of constraints.

**2. Parallel, Declarative Layout Solving.** Constraints generated by the selector matching step must be solved before a renderer can map element shapes into a grid of pixels. The output of layout solving is the sizes and positions of elements. We present the first parallel algorithm for evaluating a flow-based layout.

CSS is informally specified, aggravating use and adherence. In contrast, we separate layout specification from implementation by using attribute grammars. We provide the first declarative specification of core layout features that are not presented by the closest previous approach, CCSS [2]. Examples of analytic benefits are our proofs of layout solving termination in log time and without performing any reflow.

**3. Font handling.** We optimize use of FreeType 2 [21], a font library common to embedded systems like the iPhone.

After an overview of browser design (Section 2) and the roles of our algorithms (Section 3), we separately introduce and evaluate our algorithms (Sections 4, 5, and 6). We refer readers to our project site [15] for source code, test cases, benchmarks, and extended discussion.

## 2. BACKGROUND

Originally, web browsers were designed to render hyperlinked documents. Later, JavaScript was introduced to enable scripting of simple animations and content transitions by dynamically modifying the document. Today, AJAX applications rival their desktop counterparts. Browsers are large and complex: WebKit's JavaScript and layout engines span over 5 million lines of code.

We show the basic data flow within a browser in Figure 2 [8]. Loading an HTML page sets off a cascade of events: the page is lexed, parsed, and translated into a tree modeling the document object model (DOM). Objects referenced by URLs is fetched and added to the document. Intertwined with receiving remote resources, the page layout is incrementally solved and painted on to the screen. Script objects are loaded and processed in a blocking manner, again intertwined with the incremental layout and painting processes. For simplicity, our current algorithms assume resources are locally available and there is no scripting.

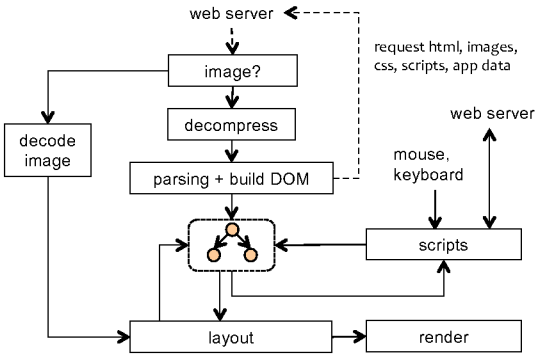



Figure 2: Data flow in a browser.

To determine optimization targets, we profiled the latest release version of Safari (4.0.3), an optimized browser. Using the Shark profiler to sample the browser's call stack every 20$\mu$s, we estimate lower bounds on CPU times when loading popular pages of the tasks shown in Figure 3. For each page, using an empty cache and a fast network, we started profiling at request time and manually stopped when the majority of content was visible. Note that, due to the call stack sampling approach, we ignore time spent idling (e.g., network time and post page load inactivity). We expect at least a magnitude of performance degradation for all tasks on mobile devices because our measurements were on a laptop that consumes about 70W under load.

We examined times for the following tasks: Flash represents the Flash virtual machine plugin, the network library handles HTTP communication (and does not include waiting on the network), parsing includes tasks like lexing CSS and generating JavaScript bytecodes, and JavaScript time represents executing JavaScript. We could not attribute all computations, but suspect much of the unclassified for samples were in layout, rendering, or CSS selector computations triggered by JavaScript, or additional tasks in creating basic HTML and CSS data structures.

| SITE \ TASK (ms) | uncategorized | CSS selectors | kernel | rendering | parsing | JavaScript | layout | network lib |
|---|---|---|---|---|---|---|---|---|
| deviantART | 384 | 119 | 224 | 102 | 171 | 65 | 29 | 20 |
| Facebook | 400 | 208 | 139 | 130 | 164 | 94 | 41 | 17 |
| Gmail | 881 | 51 | 404 | 505 | 471 | 437 | 283 | 16 |
| MSNBC | 498 | 130 | 291 | 258 | 133 | 95 | 85 | 23 |
| Netflix | 251 | 93 | 130 | 95 | 49 | 20 | 21 | 11 |
| Slashdot | 390 | 1092 | 94 | 109 | 119 | 110 | 63 | 6 |
| AV. ms | 495 | 280 | 233 | 194 | 176 | 113 | 72 | 19 |
| AV. % | 31 | 18 | 15 | 12 | 11 | 7 | 4 | 1 |

Figure 3: Task times (ms) on page load (2.4GHz laptop).

| SITE \ TASK (ms) | image rendering | text layout | glyph rendering | box layout | box rendering | CSS selectors |
|---|---|---|---|---|---|---|
| deviantART | 13 | 15 | 33 | 14 | 57 | 119 |
| Facebook | 10 | 23 | 29 | 17 | 91 | 208 |
| Gmail | 1 | 45 | 108 | 239 | 396 | 51 |
| MSNBC | 10 | 36 | 59 | 49 | 190 | 130 |
| Netflix | 20 | 8 | 16 | 13 | 60 | 93 |
| Slashdot | 2 | 42 | 39 | 21 | 68 | 1092 |
| AVERAGE ms | 18 | 30 | 54 | 54 | 159 | 253 |
| AVERAGE % | 3 | 5 | 9 | 9 | 28 | 44 |

Figure 4: Page load presentation time (ms), 2.4GHz laptop.

Our performance profile shows bottlenecks. Native library computations like parsing and layout account for at least half of the CPU time, which we are optimizing in our parallel browser. In contrast, optimizing JavaScript execution on these sites would eliminate at most 7% of the average attributed CPU time. In this paper, without even counting the unclassified operations, we present algorithms for 34% of the CPU time.

## 3. OPTIMIZED ALGORITHMS

Targeting a kernel of CSS, we redesigned the algorithms for taking a parsed representation of a page and processing it for display. In the following sections, we focus on bottlenecks in CSS selectors (18%), layout (4%), and rendering (12%). Figure 4 further breaks down these task times in Safari and presents percentages in terms of the tasks shown. Rendering is split between text, image, and box rendering. We do not present algorithms for image rendering as it can handled as a simplified form of our glyph rendering algorithm nor for box rendering as an algorithm analogous to our layout one can be used. While the figure differentiates between text and box layout, our layout algorithm treats them uniformly.

Figure 5 depicts, at a high level, the sequence of our parallel algorithms. For input, a page consists of an HTML tree of content, a set of CSS style rules that associate layout constraints with HTML nodes, and a set of font files. For output, we compute absolute element positions. Each step in the figure shows what information is computed and depicts the parallelization structure to compute it. Arrowless lines show tasks are independent while arrows describe a task that must complete before the pointed to task may compute. Generally, HTML tree elements (the nodes) correspond to tasks. Our sequence of algorithms is the following:

**Step 1 (selector matching)** determines, for every HTML node, which style constraints apply to it. For example, style rule `div a {font-size: 2em}` specifies that an "a" node descendant from a "div" node has a font size twice of its parent's. For parallelism, rules may be matched against nodes independently of each other and other nodes.

**Steps 2, 4-6 (box and text layout)** solve layout constraints. Each step is a single parallel pass over the HTML tree. Consider a node's font size, which is constrained as a concrete value or a percentage of its parent's: step 2 shows

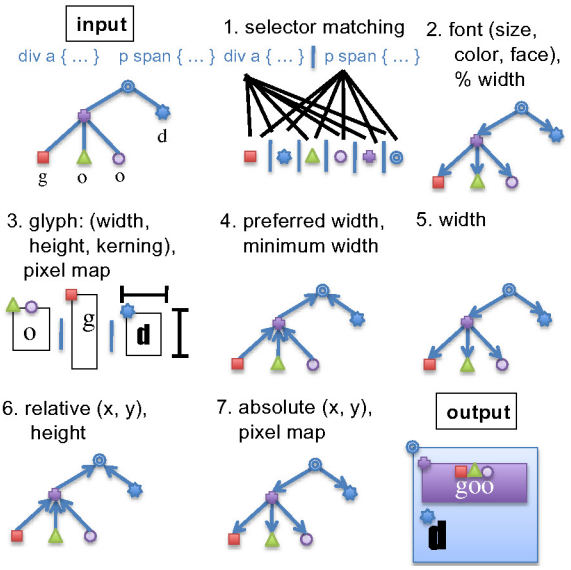



Figure 5: Parallel CSS processing steps.

that once a node's font size is known, the font size of its children may be determined in parallel. Note text is on the DOM tree's fringe while boxes are intermediate nodes.

**Step 3 (glyph handling)** determines what characters are used in a page, calls the font library to determine character constraints (e.g., size and kerning), and renders unique glyphs. Handling of one glyph is independent of handling another. Initial layout solving must first occur to determine font sizes and types. Glyph constraints generated here used later in layout steps sensitive to text size.

**Step 7 (painting or rendering)** converts each shape into a box of pixels and blends it with overlapping shapes.

We found parallelism within every step, and, in some cases, even obtained sequential speedups. While our layout process has many steps, it essentially interleaves four algorithms: a CSS selector matcher, a constraint solver for CSS-like layouts, and a glyph renderer. We can now individually examine the first three algorithms, where we achieve speedups from 3x to 80x (Figures 8, 13, and 16). Beyond the work presented here, we are applying similar techniques to related tasks between steps 1 and 2 like cascading and normalization, and GPU acceleration for step 8, painting.

## 4. CSS SELECTOR MATCHING

This section describes and evaluates our algorithm for CSS selector matching. Our innovations are in parallelization and in improving memory locality.

### 4.1 Problem Statement

Recall that CSS rules declaratively associate style constraints with document nodes. The input to CSS rule matching is a document tree and a CSS *style sheet*. A style sheet is a list of *CSS rules*. A CSS rule consists of a *selector* and a set of *style constraints* that are to be associated with the document nodes that match the selector. Consider the rule `p img { margin: 10px; }` which specifies that the margin of images in a paragraph should be 10 pixels. An `img` node is

*nested* in a paragraph if it is a descendant of a `p` node. The term `p img` is a selector and the term `{ margin: 10px; }` is the (singleton) set of style constraints. The output of rule matching is an annotation that maps each DOM node to the set of CSS rules that match this node.

When multiple rules match a node, conflicting constraints may be imposed on a node attribute (e.g., on its color). Conflicting constraints are resolved according to "cascading" rules [4]. We have not optimized this phase as it is dominated by selector matching.

*The 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 $s$. We observed that CSS rules in common use fall into a selector language that is a subset of regular expressions. This common subset of CSS covers over 99% of the rules we encountered on popular sites listed on `alexa.com`. We define our efficient algorithm for this regular subset of CSS; selectors outside this subset are handled with an unoptimized algorithm.

The regular subset of CSS selectors is defined in Figure 6(a). The operator `,` denotes disjunction: the rule matches if any of the `sel`ectors in the `rule` matches the path. A node predicate `nodePred` matches a node if the predicate's tag, id, and class attributes are a subset of the node's attributes. There is at least one tag, id, or class attribute in the predicate. For an example, the node `<div id="account" class="first,on"/>` is matched by the symbol `div.first`. The operator `s1 < s2` signifies that the node matching the selector `s1` must be the parent of the node matching the selector `s2`. The operator `s1 s2` signifies that the node matching the selector `s1` must a predecessor of the node matching the selector `s2`.

The translation of this regular subset of CSS selectors is given in Figure 6(b). The path from a node to the document root is a string, with each character representing a node. Specifically, the character represents a node matching a node predicate. The regular expression operator "`|`" stands for disjunction and "`.*`" stands for a string of arbitrary characters.

## 4.2 The CSS Matching Algorithm

Popular sites like `Slashdot.org` may have thousands of document nodes and thousands of rules to be matched against each document node. Figure 7 presents pseudocode for our selector matching algorithm, including many (but not all) of our optimizations. We one assumptions to simplify the presentation: we assume the selector language is restricted to the one defined above.

(a) Selector language

```
rule = sel | rule ","  sel
sel =
   nodePred
 | sel "<" sel
 | sel sel
nodePred =
   tag ("#"id)? (.class)*
 | "#"id ("."class)*
 | ("."class)+
```

(b) Regex subset

```
rule = sel | rule "|" sel
sel =
   symbol
 | sel sel
 | sel ".*" sel
```

Figure 6: Selector subset to regular expression translation.

```
INPUT: doc : Node Tree, rules : Rule Set
OUTPUT: nodes : Node Tree where Node =
  {id: Token, classes: Token List, tag: Token,  //input
   rules: Rule Set}                             //output

idHash, classHash, tagHash = {} //generate lookup tables
for r in rules: //redundancy elimination and hashing
  for s in rule.predicates:
    if s.last.id:
      idDash[s].map(s.last.id, r) //assume multimap is
    else if s.last.classes:       //automatically made
      classHash[s].map(s.last.classes.last, r)
    else: tagDash[s].map(s.last.tag, r)

random_parallel_for n in doc: //tile 1: ID predicates
  n.matchedList = [].preallocate(15) //locally allocate
  if n.id: attemptHashes(n, idHash, n.id)
random_parallel_for n in doc: //tile 2: class predicates
  for c in n.classes:
    attemptHashes(n, classHash, c)
random_parallel_for n in doc: //tile 3: tag predicates
  if n.tag: attemptHashes(n, tagHash, n.tag)

random_parallel_for n in doc: //reduction: determine
  for rules in n.matchedList: // rules from selectors
    for r in rules:
      n.rules.insert(r)       //note rules is a set

def attemptHashes(n, hash, idx):
  for (s, rules) in hash[idx]:
    if (matches(n, s)):       //tight right-to-left loop
      n.matchedList.push(rules) //overlapping sets
```

Figure 7: Most of our selector matching algorithm kernel.

Our algorithm first creates hash tables associating attributes with selectors that may end with them. It then, in 3 passes over the document, matches nodes against selectors. Finally, it performs a post-pass to format the results.

Some of our optimizations are adopted from WebKit:

**Hash tables.** Consider selector "`p img`": only images need to be checked against it. For every tag, class, and id instance, a preprocessor create a hash table associating attributes with the restricted set of selectors that end with it, such as associating attribute `img` with selector`p img`. Instead of checking the entire style sheet against a node, we perform the hash table lookups on its attributes and only check these restricted selectors.

**Right-to-left matching.** For a match, a selector must end with a symbol matching the node. Furthermore, most selectors can be matched by only examining a short suffix of the path to a node. By matching selectors to paths right-to-left rather than left-to-right, we exploit these two properties to achieve a form of short-circuiting in the common case.

We do not examine the known optimization of using a trie representation of the document (based on attributes). In this approaches, matches on a single node of the collapsed tree may signify matches on multiple nodes in the preimage.

We contribute the following optimizations:

**Redundant selector elimination.** Due to the weak abstraction mechanisms in the selector language, multiple rules often use the same selectors. Preprocessing avoids repeatedly checking the same selector against the same node.

**Hash Tiling.** When traversing nodes, the hash table associating attributes with selectors is randomly accessed. The HTML tree, hash table, and selectors do not fit in L1

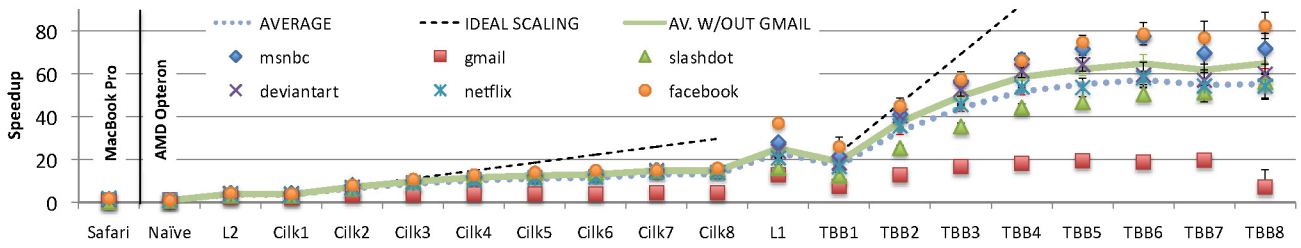



Figure 8: Selector speedup relative to a reimplementation (column 2) of Safari's algorithm (column 1). Labels Cilk<$i$> and TBB<$i$> represent the number of contexts. Column Safari on a 2.4GHz laptop, rest on a 4-core × 8-socket 2.3GHz Opteron.

cache and sometimes even L2: cache misses for them have a 10-100x penalty. We instead partition the hash table, performing a sequence of passes through the HTML tree, where each pass uses one partition (e.g., `idHash`).

**Tokenization.** Representing attributes likes tag identifiers and class names as unique integer tokens instead of as strings decreases the size of data structures (decreasing cache usage), and also shortens comparison time within the `matches` method to equating integers.

**Parallel document traversal.** Currently, we only parallelize the tree traversals. We map the tree into an array of nodes, and use a work-stealing library to allocate chunks of the array to cores. The hash tiling optimization still applies by performing a sequence of parallel traversals (one for each of the `idHash`, `classHash`, and `tagHash` hashtables).

**Random load balancing.** Determining which selectors match a node may take longer for one node than another. Neighbors in a document tree may have similar attributes and therefore the same attribute path and processing time. This similarity between neighbors means matching on different subtrees may take very different amount of times, leading to imbalance for static scheduling and excessive scheduling for dynamic approaches. Instead, we randomly assign nodes to an array and then perform work-stealing on a parallel loop, decreasing the amount of steals.

**Result pre-allocation.** Instead of calling a memory allocator to record matched selectors, we try to preallocate space. We tune on popular sites to predict space needs.

**Delayed set insertion.** The set of selectors matching a node may correspond to a much bigger set of rules because of our redundancy elimination. When recording a match, to lower memory use, we only record the selector matched, only later determining the set of corresponding rules.

**Non-STL sets.** When flattening sets of matched rules into one set, we do not use the C++ standard template library (STL) set data structure. Instead, we preallocate a vector that is the size of all the potential matches (which is an upper bound) and then add matches one by one, doing linear (but faster) collision checks.

## 4.3 Evaluation

Figure 8 reports using our rule matching algorithm on popular websites run on a 2.3 GHz 4-core × 8-socket AMD Opteron 8356 (Barcelona). Column 2 measures our reimplementation of Safari's algorithm (column 1, run on a 2.4GHz Intel Core Duo): our reimplementation was within 30% of the original and handled 99.9% of the encountered CSS rules, so it is fairly representative. GMail, as an optimization, does not significantly use CSS: we show average speedups with and without it (the following discussion of averages is without it). We performed 20 trials for each measurement. There was occasional system interference, so we dropped trials deviating over 3x (less than 1% of the trials).

We first examine low-effort optimizations. Column "L2 opts" depicts simple sequential optimizations such as the hash table tiling. This yields a 4.0x speedup. Using Cilk++, a simple 3-keyword extension of C++ for work-stealing task parallelism, we spawn selector matching tasks during the normal traversal of the HTML tree instead of just recurring. Sequential speedup dropped to 3.8x, but, compensating, strong scaling was to 3 hardware contexts with smaller gains up to 7 contexts ("Cilk" columns). Overall, speedup is 13x and 14.8x with and without Gmail.

We now examine the other sequential optimizations (Section 4.2) and changing parallelization strategy. The sequential optimizations (column "L1 opts") exhibit an average total 25.1x speedup, which is greater than the speedup from using Cilk++, but required more effort. The result of using Intel's TBB library for more verbose but efficient task parallelism and a randomized for-loop is shown in the "TBB" columns. Similiar to Cilk++, parallelization causes speedup to drop to 19x for sequential code. Strong scaling is again to 3 hardware contexts and does not plateau until 6 hardware contexts. Speedup variance increases with scaling, but less than when using the tree traversal (not shown). With and without Gmail, the speedup is 55.2x and 64.8x, respectively.

Overall, we saw total selector matching runtime dropped from an average 204ms when run on the AMD machine down to an average 3.5ms. Given an average 284ms was spent in Safari on the 2.4GHz Intel Core 2 Duo MacBook Pro, we predict unoptimized matching takes about 3s on a handheld. If the same speedup occurs on a handheld, time would drop down to about 50ms, solving the bottleneck.

# 5. LAYOUT CONSTRAINT SOLVING

Layout consumes an HTML tree where nodes have symbolic constraint attributes set by the earlier selector matching phase. Layout solving determines details like shape, text size, and position. Finally, painting converts these shapes into pixels: while we have reused our basic algorithm for a simple multicore renderer, we defer examining painting for future work that exploits data-parallel hardware like GPUs.

In a box layout, intermediate nodes represent rectangles visually nested within the rectangle of their parent and are adjacent to boxes of their sibling nodes. With a flow layout, they are also subject to constraints like word-wrapping. Text and images are fringe nodes with constraints such as

letter size or aspect ratio. To solve for one attribute, many other nodes and their attributes are involved, potentially with conflicting or cyclic relationships. It is difficult to implement layout correctly, and more so efficiently.

As with selector matching, no special annotations are required to benefit from our algorithms. Instead, we focus on finding implicit parallelism in subset of CSS. This subset is expressive: it includes the key features that developers endorse for resizable (*liquid*) layout and reveals diverging interpretations by different browsers. Ultimately, we found it simplest to define a syntax-driven transformation of CSS into a new, simpler intermediate language, which we dub Berkeley Style Sheets (BSS).

We make three contributions for layout solving:

**Performance.** We show how to decompose layout into multiple parallel passes. In Safari, the time spent solving box and text constraints is, on average, 15% of the time (84ms on a fast laptop and we expect 1s on a handheld).

**Specification.** We demonstrate a basis for the declarative specification of CSS. The CSS layout standard is informally written, cross-cutting, does not provide insight into even the naïve implementation of a correct engine, and underspecifies many features. As a result, designer productivity is limited by having to work around functionally incorrect engine implementations. Troubling, there are also standards-compliant feature implementations with functionally inconsistent interpretations between browsers. We spent significant effort in understanding, decomposing, and then recombining CSS features in a way that is more orthogonal, concise, and well-defined. As a sample benefit, we are experimenting with automatically generating a correct solver.

**Proof.** We prove layout solving is at most linear in the size of the HTML tree (and often solvable in log time). Currently, browser developers cannot even be sure that layout solving terminates. In practice, it occasionally does not [20].

Due to space constraints, we only detail BSS0, a simple layout language for vertical and horizontal boxes. It is simple enough to be described with one attribute grammar, but complicated enough that there may be long dependencies between nodes in the tree and the CSS standard does not define how it should be evaluated. We informally discuss BSS1, a multipass grammar which supports shrink-to-fit sizing, and BSS2, which supports left floats (which we believe are the most complicated and powerful elements in CSS2.1).

## 5.1 Specifying BSS0

BSS0, our simplest language kernel, is for nested layout of boxes using vertical stacking or word-wrapping. We provide an intuition for BSS0 and our use of an attribute grammar to specify it. Even for a small language, we encounter subtleties in the intended meaning of combinations of various language features and how to evaluate them.

Figure 9 illustrates the use of the various constraints in BSS0 corresponding to the output in Figure 10. The outermost box is a vertical box: its children are stacked vertically. In contrast, its second child is a horizontal box, placing its children horizontally, left-to-right, until the right boundary is reached, and then word wrapping. Width and height constraints are concrete pixel sizes or percentages of the parent. Heights may also be set to `auto`: the height of the horizontal box is just small enough to contain all of its children. BSS1 [15] shows extending this notion to width calculations adds additional but unsurprising complexity.

$$
\begin{aligned}
&VBOX[wCnstrnt = 200px, hCnstrnt = 150px](\\
&\quad VBOX[wCnstrnt = 80\%, hCnstrnt = 15\%](),\\
&\quad HBOX[wCnstrnt = 100px, hCnstrnt = auto](\\
&\quad\quad VBOX[wCnstrnt = 40px, hCnsntrt = 15px](),\\
&\quad\quad VBOX[wCnstrnt = 20px, hCnstrnt = 15px](),\\
&\quad\quad VBOX[wCnstrnt = 80px, hCnstrnt = 15px]()))
\end{aligned}
$$

Figure 9: Sample BSS0 layout constraints input.

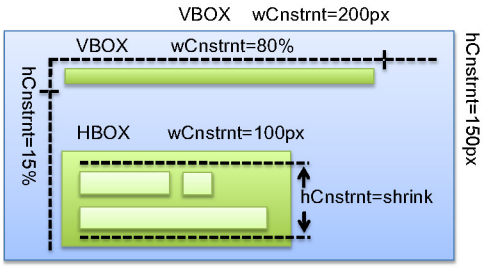



Figure 10: Sample BSS0 layout constraints output.

We specify the constraints of BSS0 with an attribute grammar (Figure 11). The goal is, for every node, to determine the width and height of the node and its x and y position relative to its parent. The bottom of the figure defines the types of the constraints and classes V and H specify, for vertical and horizontal boxes, the meaning of the constraints.

In an attribute grammar [11], attributes on each node are solved during tree traversals. An inherited attribute is directly dependent upon attributes of its parent node, such as a width being a percentage of its parent width's. A synthesized attribute is directly dependent upon attributes of its children. For example, if a height is set to `auto` – the sum of the heights of its children – we can solve them all in an upwards pass. Both inherited and synthesized attributes may be functions of both inherited and synthesized attributes. In unrestricted attribute grammars, indirect dependencies for an attribute may be both above and below in the tree: a traversal may need to repeatedly visit the same node, potentially with non-deterministic or fixed-point semantics!

BSS0 is designed such that inherited attributes are functions of inherited attributes: a traversal to solve them need only observe a partial order going downwards in the tree. Topological, branch-and-bound, and depth-first traversals satisfy it. Similarly, synthesized attributes, except on the fringe, only depend on synthesized attributes: after inherited attributes are computed, a topologically upwards traversal may compute the synthesized ones in one pass. In Section 5.3, we see this simplifies parallelization: parallel downwards and then upwards passes suffice for BSS0 (steps 2 and 4 of Figure 5). In the node interface (Figure 11), we annotate attributes with dependency type (inherited or synthesized).

In our larger languages, [15] inherited attributes may also access synthesized attributes: two passes no longer suffice. In these extensions, inherited attributes in the grammar are separated by an equivalence relation, as are synthesized ones, and the various classes are totally ordered: each

```
interface Node                          // passes
  @input children, prev, wCnstrnt, hCnstrnt
  @grammar1:            // (top-down, bottom-up)
    @inherit width      // final width
    @inherit th         // temp height for bad constraint
    @inherit relx       // x position relative to parent
    @synthesize height // final height
    @synthesize rely    // y position relative to parent

class V implements Node                 // semantic actions
  @grammar1.inherit    // top-down
  for c in children:
    c.th = sizeS(th, c.hCnstrnt) //might be auto
    c.width = sizeS(width, c.wCnstrnt)
    c.relx = 0

  @grammar1.synthesize // bottom-up
  height = joinS(th, sum([c.height | c in children]))
  if children[0]: children[0]rely = 0
  for c > 0 in children:
    c.rely = c.prev.rely + c.prev.height

class H implements Node                 // semantic actions
  @grammar1.inherit    // top-down
  for c in children:
    c.th = sizeS(th, c.hCnstrnt) //might be auto
    c.width = sizeS(width, c.wCnstrnt)
  if children[0]:
    children[0]relx = 0
  for c > 0 in children:
    c.relx = c.prev.relx + c.prev.width > width ? // wrap
      0 : c.prev.relx + c.prev.width

  @grammar1.synthesize // bottom-up
  if children[0]:
    children[0]rely = 0
  for c > 0 in children:
    c.rely = c.prev.relx + c.prev.width > width ? // wrap
      c.prev.rely + c.prev.height : c.prev.rely
  height =
    joinS(th, max([c.rely + c.height | c in children]))

class Root constrains V // V node with hardcoded values
  th = 100              // browser specifies all of these
  width = 100, height = 100
  relx = 0, rely = 0

function sizeS (auto, p %) -> auto   // helpers
            | (v px, p %) -> v * 0.01 * p px
            | (v, p px) -> p px
            | (v, auto) -> auto
function joinS (auto, v) -> v
            | (p px, v) -> p
```

$$R \to V \mid H \qquad \text{// types}$$
$$V \to H^* \mid V^*$$
$$H \to V^*$$

$$\begin{aligned}
V ::& \{\text{wCnstrnt} : P \mid PCNT,\ \text{hCnstrnt} : P \mid PCNT \mid \text{auto}\\
&\text{children} : V\ list,\ \text{prev} : V,\\
&\text{th} : P \mid \text{auto},\\
&\text{width} = P,\ \text{relx} : P,\ \text{rely} : P,\ \text{height} : P\}\\
H ::& \{\text{wCnstrnt} : P \mid PCNT,\ \text{hCnstrnt} : P \mid PCNT \mid \text{auto}\\
&\text{children} : V\ list,\ \text{prev} : V,\\
&\text{th} : P \mid \text{auto},\\
&\text{width} = P,\ \text{relx} : P,\ \text{rely} : P,\ \text{height} : P\}\\
Root ::& V \text{ where } \{\text{width} : P,\ \text{height} : P,\ \text{th} : P\}\\
P ::& \mathbb{R} \text{ px}\\
PCNT ::& \mathbb{P}\ \% \text{ where } \mathbb{P} = [0,1] \subset \mathbb{R}
\end{aligned}$$

Figure 11: BSS0 passes, constraints, helpers, and structure.

class corresponds to a pass. All dependency graphs of attribute constraints abide by this order. Alternations between sequences of inherited and synthesized attributes correspond to alternations between upwards and downwards passes, with the total amount of passes being the number of equivalence classes. Figure 5 shows these passes. The ordering is for the pass by which a value is definitely computable (which our algorithms make a requirement); as seen with the relative x coordinate of children of vertical nodes, there are often opportunities to compute in earlier passes.

## 5.2 Surprising and Ambiguous Constraints

Even for a seemingly simple language like BSS0, we see scenarios where constraints have a surprising or even undefined interpretation in the CSS standard and browser implementations. Consider the following boxes:

*V[hCnstrnt=auto](V[hCnstrnt=50%](V[hCnstrnt=20px]))*

Defining the height constraints for the outer 2 vertical boxes based on their names, the consistent solution would be to set both heights to 0. Another approach is to ignore the percentage constraint and reinterpret it as `auto`. The innermost box size is now used: all boxes have height 20px. In CSS, an analogous situation occurs for widths. The standard does not specify what to do; instead of using the first approach, our solution uses the latter (as most browsers do).

Another subtlety is that the width and height of a box does not restrict its children from being displayed outside of its boundaries. Consider the following:

*V[hCnstrnt=50px](V[hCnstrnt=100px])*

Instead of considering such a layout to be inconsistent and rejecting it, BSS0 (like CSS) accepts both constraints. Layout proceeds as if the outer box really did successfully contain all of its children. Depending on rendering settings, the overflowing parts of the inner box might still be displayed.

We found many such scenarios where the standard is undefined, explicitly or possibly by accident. In contrast, our specification is well-defined.

## 5.3 Parallelization

Attribute grammars expose opportunities for parallelization [9]. First, consider inherited attributes. Data dependencies flow down the tree: given the inherited attributes of a parent node, the inherited attributes of its children may be independently computed. Second, consider synthesized attributes: a node's childrens' attributes may be computed independently. Using the document tree as a task-dependency graph, arrows between inherited attributes go downwards, synthesized attribute dependencies upwards, and the fringe shows synthesized attributes are dependent upon inherited attributes from the previous phase (Figure 5).

Many parallel algorithms are now possible. For example, synthesized attributes might be computed with prefix scan operations. While such specialized operators may support layout subsets, we found much of the layout time in Safari is spent in more general operations (e.g., $isSVG()$). We instead take a task-parallel approach (Figure 12). For each node type and grammar, we define general (sequential) functions for computing inherited attributes (*calcInherited()*) and synthesized attributes (*calcSynthesized()*). Scheduling is orthogonally handled as follows:

We define parallel traversal functions that invoke layout calculation functions (*semantic actions* [11]). One grammar is fully processed before the next. To process a grammar,

```
class Node
  def traverse (self, g):
    self['calcInherited' + g]();
    @autotune(c.numChildren) //sequential near fringe
    parallel_for c in self.children:
      c.traverse(g) //in parallel to other children
    self['calcSynthesized' + g]();
class V: Node
  def calcInheritedG1 (self):
    for c in self.children:
      c.th = sizeS(self.th, c.hCnstrnt)
      c.width = sizeS(self.tw, c.wCnstrnt)
  def calcSynthesizedG1 (self):
    self.height =
      joinS(self.th,
            sum([c.height where c in self.children]))
    if self.children[0]: self.children[0].rely = 0
    for c > 0 in sel.children:
      c.rely = c.prev.rely + c.prev.height
    self.prefWidth =
      join(self.tw,
           max([c.prefWidth where c in self.children]))
    self.minWidth =
      join(self.tw,
           max([c.minWidth where c in self.children]))
   ...
...

for g in ['G1', ... ]: //compute layout
  rootNode.traverse(g)
```

Figure 12: BSS0 parallelization pseudocode. Layout calculations are implemented separately from the scheduling and synchronization traversal function.

a recursive traversal through the tree occurs: inherited attributes are computed for a node, tasks are spawned for processing child nodes, and upon their completion, the node's synthesized attributes are processed. Our implementation uses Intel's TBB, a task parallel library for C++. Traditional optimizations apply, such as tuning for when to sequentially process subtrees near the bottom of the HTML tree instead of spawning new tasks. Grammar writers define sequential functions to compute the attributes specified in Figure 11 given the attributes in the previous stages; they do not handle concerns like scheduling or synchronization.

## 5.4 Performance Evaluation

Encoding a snapshot of `slashdot.org` with BSS1, we found that box layout time takes only 1-2ms with another 5ms for text layout. In contrast, our profile of Safari attributes 21ms and 42ms, respectively (Figure 4). We parallelized our implementation, seeing 2-3x speedups (for text; boxes were too fast). We surmise our grammars are too simple. We then performed a simple experiment: given a tree with as many nodes as Slashdot, what if we performed multiple passes as in our algorithm, except uniformly spun on each node so that the total work equals that of Slashdot, simulating the workload in Safari? Figure 13 shows, without trying to optimize the computation any further and using the relatively slow but simple Cilk++ primitives, we strongly scale to 3 cores and gain an overall 4x speedup. The takeaway is that our algorithm exposes exploitable parallelism; as our engine grows, we will be able to tune it as we did with selectors.

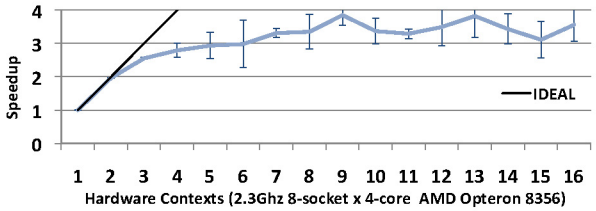
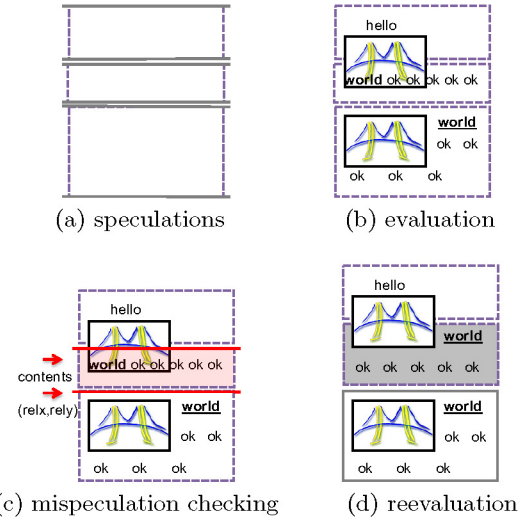



Figure 13: Simulated layout parallelization speedup.




(a) speculations (b) evaluation

(c) mispeculation checking (d) reevaluation

Figure 14: Speculative evaluation for floats.

## 5.5 Floats

Advanced layouts such as multiple columns or liquid (multiresolution) flows employ *floating* elements. For example, Figure 14(d) depicts a newspaper technique where images float left and content flows around them. Floats are ambiguously defined and inconsistently implemented between browsers; our specification of them took significant effort and only supports left (not right) floats. We refer to our extended version [15] for more detailed discussion.

A revealed weakness of our approach in BSS0 and BSS1 is that floating elements may have long sequential dependencies. For example, attempting to solve the second paragraph in Figure 14(b) without knowing the positions of elements in the first leads to an incorrect layout. Our solution is to speculatively evaluate grammars (Figures 14(a), (b)) and then check for errors (Figure 14(c)), rerunning grammars that misspeculate (Figure 14(d)). Our specification-driven approach makes it clear which values need to be checked.

## 5.6 Termination and Complexity

Infinite loops occasionally occur when laying out webpages [20]. Such behavior might not be an implementation bug: there is no proof that CSS terminates! Our specification approach enables proof of a variety of desirable properties. At hand are termination and asymptotic complexity.

We syntactically prove for BSS0 that layout solving terminates, computes in time at worst linear in HTML tree size, and for a large class of layouts, computes in time log

of HTML tree size. Our computations are defined as an attribute grammar. The grammar has an inherited computation phase (which is syntactically checkable): performing it is at worst linear using a topological traversal of the tree. For balanced trees, the traversal may be performed in parallel by spawning at nodes: given $log(|tree|)$ processors, the computation may be performed in log time. A similar argument follows for the synthesized attribute pass, so these results apply to BSS0 overall. A corollary is that reflow (iterative solving for the same attribute) is unnecessary.

Our extended version [15] discusses extending these techniques to richer layout languages. An exemption is that we cannot prove log-time speculative handling of floats.

## 6. FONT HANDLING

Font library time, such as for glyph rendering, takes at least 10% of the processing time in Safari (Figure 4). Calls into font libraries typically occur greedily whenever text is encountered during a traversal of the HTML tree. For example, to process the word "good" in Figure 14, calls for the bitmaps and size constraints of 'g', 'o', and 'o' would be made at one point, and, later, for 'd'. A cache is used to optimize the repeated use of 'o'.

Figure 15 illustrates our algorithm for handling font calls in bulk. This is step 3 of our overall algorithm (Figure 5): it occurs after desired font sizes are known for text and must occur before the rest of the layout calculations (e.g., for *prefWidth*) may occur. First, we create a set of necessary font library requests – the smallest abstract request is the combination of *(character, font face, size,* and *style*) – and then organize and perform parallel calls for these requests. Our algorithm currently perform the pooling step sequentially, but there might be exploitable concurrency as it can be described as a parallel reduction that takes the unions of sets. Finally, our algorithm makes nested `parallel_for` work-stealing calls using TBB, hierarchically encoding affinity based on font file and creating tasks at the granularity of (*font, size*).

Figure 16 shows the performance of our algorithm on several popular sites. We use the FreeType2 font library[21], Intel's TBB for a work stealing `parallel_for`, and a 2.66GHz Intel Nehalem with 4 cores per socket. For each site, we extract the HTML tree and already computed font styles (e.g., bold) as input for our algorithm. We see strong parallelization benefits for up to 3 cores and a plateau at 5. In an early implementation, we also saw about a 2x sequential speedup, we guess due to locality benefits from staging instead of greedily calling the font library. Finally, we note

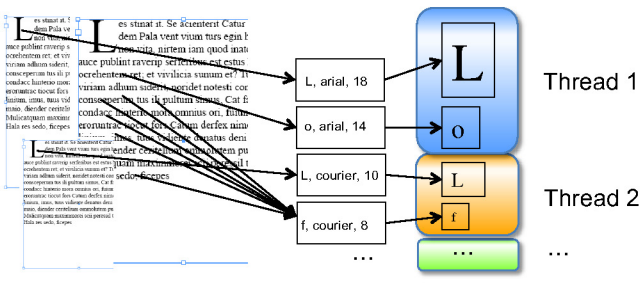



Figure 15: Bulk and parallel font handling.

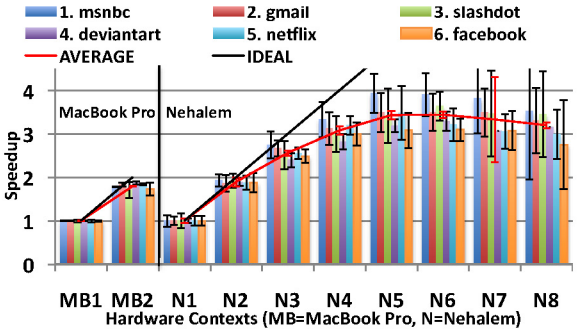



Figure 16: Glyph rendering parallelization speedup. Column label shows number of hardware contexts used.

the emergence of Amdahl's Law: before parallelization, our sequential processor to determine necessary font calls took only 10% of the time later spent making calls, but, after optimizing elsewhere, it takes 30%. Our font kernel parallelization succeeded on all sites with a 3-4x speedup.

## 7. RELATED WORK

**Multi-process browsers.** Browsers use processes to isolate pages, hardening against attacks and fairly apportioning resources [22]. Such process-based browser architecture allows pipeline-style parallelism between components but does not parallelize the layout computation which remains confined into a component.

**CSS Selectors.** Our selector matching algorithm incorporates two sequential optimizations from WebKit. Using a similar base algorithm to that of WebKit, Haghighat *et al* [6] speculatively parallelize the task of matching one selector against one node, achieving two-fold speedups. Our results suggest that executing this task sequentially, but with many instances in parallel, leads to higher speedups.

Bordawekar *et al* [3] study matching XPath expressions against XML trees. They experiment with *data partitioning* (spreading the tree across multiple processors and concurrently matching the full query against the partitions) and *query partitioning* (partitioning the parameter space of a query across processors). Their problem is biased towards single queries and large files while ours is the opposite. We partition data by randomly distributing tree leaves, partition queries by splitting disjunctions, and we introduce (temporal) tiling of the set of selectors.

**Glyph rendering.** Parallelism has been previously proposed for rendering an individual glyph [17]. We divide initial glyph rendering from compositing, lessoning this need. Furthermore, we use the layout engine to schedule concurrent glyph renderings and can reuse sequential glyph rendering engines.

**Specifying layout.** Most document layout systems, like TEX and CSS, have informal specifications [12, 4], if any. For performance reasons, they are typically implemented in low-level languages. Increasingly, high-level languages like ActionScript are used, as in Adobe Flex, but that does not sufficiently elevate the level of implementation abstraction. Executable declarative specification of flexible document layouts is a difficult problem. Heckmann et al [7] provide an implementation of LATEX formula layout in functional subset of SML. The Cassowary project [2] shows how to model

cascading, inheritance, and a simplified view of tables using linear and finite-domain constraints. Unfortunately, the rest of the box model is handled by a native CSS engine. Swierstra *et al* [19] show how to use attribute grammars to encode a simplification of the fixed (non-default) HTML table algorithm. This task is similar to the table model supported by Cassowary. To encode a basic box model extended with word-wrapping but no features like floats, Lin [13] proposes a pipeline of linear solvers mixed with native engines. In contrast, we provide a declarative, executable specification with attribute grammars for a representative set of CSS-like features and without external engines. Our performance simulation seems promising (see Section 5) and we have demonstrated analytic benefits of our restricted modeling language.

**Attribute grammars.** Attribute grammars are a well-studied model [11]. They have primarily been examined as a language for the specification and implementation of compilers. Parallel evaluation of attribute grammars is well understood. Jourdan provides a survey [9] of techniques for finding and exploiting parallelism. We contribute a layout encoding for CSS that is parallelizable and the technique of speculative attributes to improve parallelization.

# 8. CONCLUSION

We have demonstrated algorithms for three bottlenecks of loading a webpage: matching CSS selectors, laying out general elements, and text processing. Our sequential optimizations feature improved data locality and lower cache usage. Browsers are seeing increasingly little benefit from hardware advances; our parallel algorithms show how to take advantage of advances in multicore architectures. We believe such work is critical for the rise of the mobile web.

Our specification of layout as attribute grammars is of further interest. We have proved that, not only does layout terminate, but it is possible without reflow and often in log time. We expect further benefits from achieving a declarative specification of the core of a layout system.

Overall, our approach simplifies tasks for browser developers and web designers dependent upon them. This work is a milestone in our construction of a parallel, mobile browser for browsing the web on 1 Watt.

# 9. SUPPORT

Krste Asanovic, Chan Siu Man, Chan Siu On, Chris Jones, Robert O'Callahan, Heidi Pan, Ben Hindman, Rajesh Nishtala, Shoaib Kamil, and Andrew Waterman have provided valuable help through various stages of this work.

# 10. REFERENCES

[1] Apple, Inc. *iPhone Dev Center: Creating an iPhone Application*, June 2009.

[2] G. J. Badros. *Extending Interactive Graphical Applications with Constraints*. PhD thesis, University of Washington, 2000. Chair-Borning, Alan.

[3] R. Bordawekar, L. Lim, and O. Shmueli. Parallelization of XPath Queries using Multi-Core Processors: Challenges and Experiences. In *EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology*, pages 180–191, New York, NY, USA, 2009. ACM.

[4] B. Bos, H. W. Lie, C. Lilley, and I. Jacobs. Cascading Style sheets, Level 2 CSS2 Specification, 1998.

[5] S. Dubey. AJAX Performance Measurement Methodology for Internet Explorer 8 Beta 2. *CODE Magazine*, 5(3):53–55, 2008.

[6] M. Haghighat. Bug 520942 - Parallelism Opportunities in CSS Selector Matching, October 2009. https://bugzilla.mozilla.org/show_bug.cgi?id=520942.

[7] R. Heckmann and R. Wilhelm. A functional description of tex's formula layout. *J. Funct. Program.*, 7(5):451–485, 1997.

[8] C. Jones, R. Liu, L. Meyerovich, K. Asanovic, and R. Bodik. Parallelizing the web browser, 2009.

[9] M. Jourdan. A Survey of Parallel Attribute Evaluation Methods. In *Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 234–255. Springer Berlin / Heidelberg, 1991.

[10] N. Kerris and T. Neumayr. Apple App Store Downloads Top Two Billion. September 2009.

[11] D. E. Knuth. Semantics of Context-Free Languages. *Theory of Computing Systems*, 2(2):127–145, June 1968.

[12] H. W. Lie. *Cascading Style Sheets*. Doctor of Philosophy, University of Oslo, 2006.

[13] X. Lin. Active Layout Engine: Algorithms and Applications in Variable Data Printing. *Computer-Aided Design*, 38(5):444–456, 2006.

[14] M. Mayer. Google I/O Keynote: Imagination, Immediacy, and Innovation... and a little glimpse under the hood at Google. June 2008.

[15] L. Meyerovich. A Parallel Web Browser. `http://www.eecs.berkeley.edu/~lmeyerov/projects/pbrowser/`.

[16] L. Meyerovich. Rethinking Browser Performance. *Login*, 34(4):14–20, August 2009.

[17] J. L. Recker, G. B. Beretta, and I.-J. Lin. Font rendering on a gpu-based raster image processor. Technical Report 181, HP Laboratories, August 2009.

[18] C. Stockwell. IE8 What is Coming. http://en.oreilly.com/velocity2008/public/schedule/detail/3290, June 2008.

[19] S. D. Swierstra, P. R. A. Alcocer, J. Saraiva, D. Swierstra, P. Azero, and J. Saraiva. Designing and Implementing Combinator Languages. In *Third Summer School on Advanced Functional Programming, volume 1608 of LNCS*, pages 150–206. Springer-Verlag, 1999.

[20] G. Talbot. Confirm a CSS Bug in IE 7 (infinite loop). http://bytes.com/topic/html-css/answers/615102-confirm-serious-css-bug-ie-7-infinite-loop, March 2007.

[21] D. Turner. *The Design of FreeType2*. The FreeType Development Team, 2008. http://www.freetype.org/freetype2/docs/design/.

[22] 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 *18th Usenix Security Symposium*, 2009.