# Fast and Parallel Webpage Layout*

Leo A. Meyerovich †
UC Berkeley
lmeyerov@eecs.berkeley.edu

Rastislav Bodík
UC Berkeley
bodik@eecs.berkeley.edu

## 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. Current browsers, however, barely exploit hardware parallelism, so we are designing a parallel mobile browser. 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 layout solving 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

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

## 1. INTRODUCTION

Web browsers should be at least a magnitude faster. Current browser performance is insufficient, so companies like Google manually optimize typical pages [13] and rewrite them in low-level platforms for mobile devices[1]. As we have previously noted, browsers are increasingly CPU-bound [8, 15]. Benchmarks of Internet Explorer [16] and Safari reveal 40-70% of the average processing time is spent on visual layout, so we present our new components for layout. Crucial to exploiting modern and 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 that have been downloaded over 2,000,000,000 times by their 50,000,000+ owners [9]. 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 the low-level instead of losing 1-2 magnitudes of performance. Furthermore, these applications represent less than 1% of online content. As we consider successively smaller computing classes, our performance concerns compound. By optimizing browsers, we can make high-level platforms like the web more viable for mobile devices.

Our second motivation to optimize browsers is for 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 [13]. Other teams have confirmed these findings throughout Facebook and Google. Improving clientside 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 enable developers to instead focus more on application domain concerns.

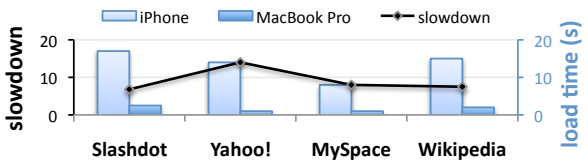



Figure 1: 400Mhz iPhone vs. 2.4Ghz MacBook Pro loadtimes using the same wireless network.

Webpage processing is replacing networking as the performance bottleneck. Figure 1 compares loadtimes for popular

*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.

websites on a 2.4 Ghz MacBook Pro to using a 400Mhz iPhone. We used the same wireless network for the tests: loadtime is still 9x slower on the handheld, suggesting the network is not entirely to blame. The IE8 team, on undisclosed hardware, found the top 30 most popular sites use an average 1.0s of CPU time [6] and our above sample shows about 1.6s. 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. Given network advances, browsers are increasingly CPU-bound.

To improve browser performance, we should exploit parallelism. Driven by Moore's Law, Proebstring's Law empirically observes that hardware improvements have far outpaced compiler-driven ones. An effective optimization strategy has therefore been to not optimize software but wait for more hardware. Unfortunately, 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 manafactured 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 at least 40% of the time in Safari is spent in these tasks and others report 70% of the time in Internet Explorer for just a subset of them [6]. We examine the following for CSS (Cascading Style Sheets [5, 11]) layout tasks:

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

**2. Layout Solving.** Constraints generated by the selector matching step must be solved before a renderer can map element shapes into a grid of pixels. CSS layout is a flow-based layout language, which is common to document systems. Focusing on a simplified kernel language, we present the first parallel algorithm for evaluating a flow-based layout. Furthermore, complicating CSS layout use and implementation, the standard is informal: in contrast, we phrase our algorithm with attribute grammars. Finally, this approach yields the first proofs of not only termination but solving in log time and without reflow.

**3. Font handling.** We optimize use of FreeType 2 [18], a font library common to embeded systems like the iPhone.

Our ultimate goal is a fast 1 Watt browser. 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). Due to space constraints, we refer readers to our project site [14] 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 providing both layout and JavaScript engines for many systems, is over 5 million lines of code.

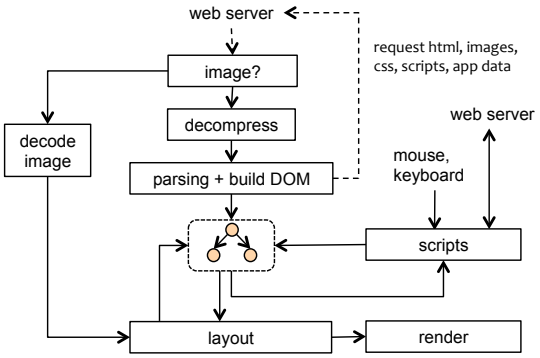



Figure 2: Data flow in a browser.

The simplified flow of data in a browser is shown in Figure 2. Loading an HTML page sets off a cascade of events: the page is scanned, parsed, and compiled into a document object model (DOM), an abstract syntax tree of the document. Content referenced by URLs is fetched and added to the DOM tree. As the content necessary to display the page becomes available, the page layout is (incrementally) solved and drawn to the screen. After the initial page load, scripts respond to events generated by user input and server messages, typically modifying the DOM. This may, in turn, cause the page layout to be recomputed and redrawn.

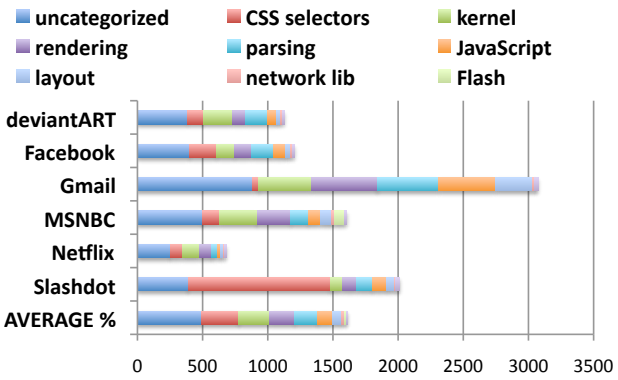



Figure 3: Task times (ms), 2.4GHz MacBook Pro.

To determine hotspots, we profiled the latest release version of Safari (4.0.3), a browser known for performance. Using the Shark profiler to sample the browser's callstack every 20$\mu$s, we estimate lowerbounds on particular task CPU times when loading popular pages (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 callstack sampling approach, we throw out time spent idling (e.g., network time and post page load inactivity). We expect at least a magnitude of performance degredation 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 wait-

ing 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 unaccounted for samples were in layout, rendering, or CSS selector computations triggered by JavaScript, or additional tasks in creating basic HTML and CSS data structures.

Our performance profile illustrates what is and is not a bottleneck. For example, optimizing JavaScript execution on these sites would eliminate at most 7% of the average CPU time. Surprisingly, more time is spent on parsing related tasks for JavaScript rather than actually running it. Native library computations account for at least half of the CPU time, which we are optimizing in our parallel browser.

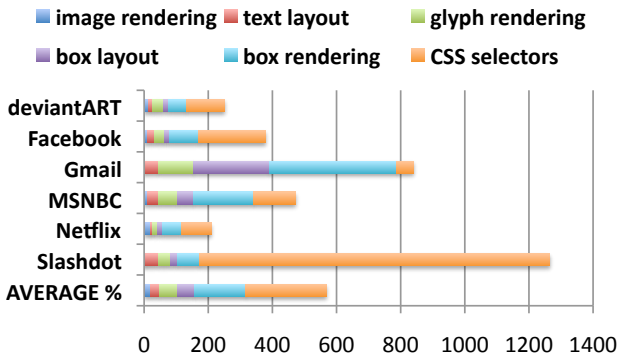



Figure 4: Presentation time (ms), 2.4GHz MacBook Pro.

In the following sections, we focus on bottlenecks in CSS selectors, layout, and text handling. Benchmarks of Internet Explorer 8 [16] by others for popular sites attribute 70% of the time to just layout and rendering. We only attribute 34% (+/- 15%) of the time to these tasks; if most of the unknown time is distributed between them (which is plausible), our benchmarks would be consistent. Under either intepretation, these tasks are expensive and CPU-bound.

## 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. Figure 4 further breaks down task times in Safari for this process. CSS selector matching combines a style template with a HTML tree of content in order to generate the set of style constraints for every individual HTML node. Box layout and text layout solve these constraints for various node types, such as determining the size of a font for a pargraph or the position of a square. The rendering tasks generate raster (pixel) images from these solved node constraints and combine them together. We optimized CSS selector matching, box and text layout, and glyph rendering.

Figure 5 depicts, at a high level, the basic 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. Blue

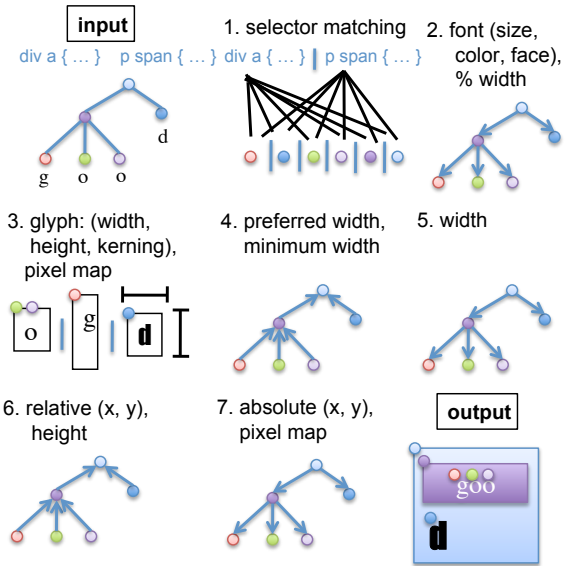



Figure 5: Parallel CSS processing steps.

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. E.g., 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 are independently matched against nodes.

**Steps 2, 4-7 (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, 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.

We found parallelism within every step above, and, in many cases, even 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 7, 11, and 14). 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

Our first algorithm optimizes the CSS selector language. Selectors declaratively associate style constraints with con-

tent, enabling designers, for example, to define global style templates. Some sites, like `google.com`, avoid runtime costs from selectors by not using them, while others, like `Zimbra` and `Slashdot`, spend 30% and 54% of their CPU time in processing selectors, respectively. We found selector matching time to be 22% in Safari and others report 10% of the time for Internet Explorer 8. [16] Our innovations are in parallelization and lowering memory pressure, and, by Amdahl's law, are successful enough to make pre- and post-processing matched selectors the new bottlenecks.

## 4.1 Problem Statement

Consider the rule `p img { margin: 10px; }` specifying that images descendant from paragraph nodes in a document tree should have a large margin. The term `p img` is a *selector*: a selector language is used to associate style constraints like `margin: 10px` with document contents. A style sheet has many such rules. A *rule matcher* determines which style constraints apply to which elements in a document. A popular site like `Slashdot.org` has thousands of nodes and thousands of selectors to match against each of those nodes: rule matching is a known optimization target.

The input to rule matching is a set of rules in the above form and a document in the form of a tree. Every rule consists of a predicate and a style constraint: if a node in the document satisfies the predicate, the style constraint applies to the node. The output of rule matching is an annotation on every node describing the set of rules applicable to that node. Multiple rules may apply to the same node, and, in practice, often do. For every node, the constraints associated with its rules must be combined (with special handling if there is a redefinition); we have not optimized this phase.

## 4.2 Selectors

We first focus on what it means for one rule to match one document node. A rule's selector is matched against the path from the node to the root of the document tree, which is a problem similar to matching a restricted form of regular expressions against strings. For the most commonly used subset of CSS (representing over 99% of the rules we encountered on popular sites like those listed on `alexa.com`), the selector language is an exact subset of regular expression. Our algorithm does not perform the following translation, but it is useful for understanding the matching problem:

(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
```

To ground the mapping to regular expression matching, we represent the path from a node to the document's root as a string where every character represents a node's attributes. A subset of the typical regular expression operators – concatenation, "|" (disjunction), and ".*" (concatenation with an arbitrary intermediate string[1]) – are used to build a rule. If one of the `sel`ectors in a `rule` matches the path, the rule matches. A `symbol` in a predicate does not have to describe all of the attributes in a node for it to match. For example, the node `<div id="account" class="first,on"/>` is matched by the symbol `div.first`. A document node matches a predicate symbol if the node contains *at least* the attributes required by the symbol.

[1]The restriction of the Kleene star to the ".*" form prevents terms like "a*", simplifying backwards matching.

```
INPUT: document : 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 = {}
for r in rules: //redundancy elimination and hashing
  for s in rule.predicates:
    if s.last.id: inject(idHash, s.last.id, s, r)
    else if s.last.classes:
      inject(classHash, s.last.classes.last, s, r)
    else: inject(tagHash, s.last.tag, s, r)

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

random_parallel_for n in document: //reduction
  for rules in n.matchedList:
    for r in rules:
      n.rules.push(r)

def inject(h, idx, s, r):
  if !h[idx]: h[idx] = multimap()
  h[idx].map(s, r)

def attemptHashes(n, hash, idx):
  for (s, rules) in hash[idx]:
    if (matches(n, s)): //a tight selector-matching loop
      n.matchedList.push(rules) //overlapping list of sets
```

Figure 6: Most of our selector matching algorithm kernel.

## 4.3 High-Level Algorithm

Figure 6 presents pseudocode for our selector matching algorithm, including many of our optimizations. We make two assumptions to simplify the presentation: we assume the selector language is restricted to the one defined above and that disjunctions are split into separate selectors.

Our algorithm first creates hashtables 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:

## 4.4 Optimizations

Some of our optimizations are adopted from WebKit:

**Hashtables.** Consider selector "`p img`": only images need to be checked against it. For every tag, class, and id instance, a preprocessor create a hashtable 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 stylesheet against a node, we perform the hashtable 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 hashtable associating attributes with selectors is randomly accessed. The HTML tree, hashtable, and selectors do not fit in L1 cache and sometimes even L2: cache misses for them have a 10-100x penalty. We instead partition the hashtable, 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 preallocate space (and, in the rare case it is exhausted, only then call the allocator). Based on samples of webpages, we preallocate spaces for 15 matches. This is tunable.

**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 upperbound) and then add matches one by one, doing linear (but faster) collision checks.

## 4.5 Evaluation

Figure 7 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 hashtable 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. Seqential 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 4.4) and changing parallelization strategy. The sequential optimzations (column "L1 opts") exhibit an average total 25.1x speedup, which is greater than the speedup from using Cilk++, but required more effort. Switching to Intel's TBB library for more verbose but lower footprint task parallelism and using a randomized for-loop is depicted in the "TBB" columns. As with Cilk++, parallelization causes speedup to drop to 19x in the sequential case, with strong scaling again to 3 hardware contexts that 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 measured 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 and text size and position. A subsequent step, painting (or rendering), converts these shapes into pixels: while we have reused our basic algorithm for a simple multicore renderer, we defer examination for future work that investigates the use of data-parallel hardware like GPUs.

For intuition, intermediate nodes represent rectangles visually nested within the rectangle of their parent (a box layout) and are adjacent to boxes of their sibling nodes. They are subject to constraints like word-wrapping (flow layout constraints). Text and images are on the tree's fringe and have constraints such as letter size or aspect ratio. To solve for one attribute, many other nodes and their attributes are involved. For example, to determine the width of a node, we must consider width constraints of descendant nodes, which might depend upon text size, which might be a function of the current node's text size, etc. It is difficult to implement layout correctly, and more so efficiently.

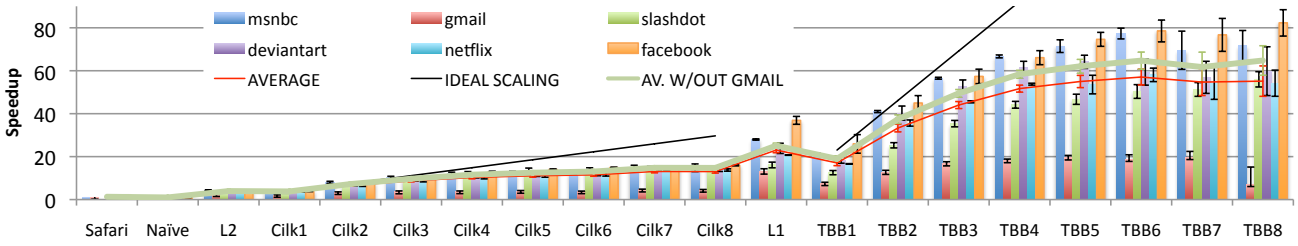



Figure 7: Selector matching speedup relative to a reimplementation (column 2) of Safari's algorithm (column 1).

As with selector matching, we do not ask that developers make special annotations to benefit from our algorithms. Instead, we focus on a subset of CSS that is large enough to reveal implemenation bugs in all mainstream browsers yet is small enough to show how to exploit parallelism. This subset is expressive: it includes the key features that developers endorse for resizable (*liquid*) layout. 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 naive 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 [17].

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 CSS0).

## 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. It corresponds to the following input:

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

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 [14] shows extending this notion to width calculations adds additional but unsurprising complexity.

We specify the constraints of BSS0 with an attribute grammar (Figure 8). 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 [10], attributes on each node are solved during tree traversals. An inherited attribute is dependent upon attributes of nodes above it in the tree, such as a width being a percentage of its parent width's. A synthesized attribute is dependent upon attributes of nodes below it. 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. An inherited attribute may be a function of both inherited and synthesized attributes, and a synthesized attribute may also be a function of both inherited and synthesized attributes. In general attribute grammars, a traversal may need to repeatedly visit the same node, potentially with non-deterministic or fixed-point semantics!

BSS0 has the useful property that inherited attributes are only functions of other 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 all do this. Similarly, synthesized attributes, except on the fringe, only depend upon other synthesized attributes: after inherited attributes are computed, a topologically upwards traversal may compute the synthesized ones in one pass. In the node interface (Figure 8), we annotate attributes with dependency type (inherited or synthesized). In Section 5.3, we see this simplifies parallelization. By design, a downwards and then upwards pass

```
interface Node                          // passes
  @input children, prev, wCnstrnt, hCnstrnt
  @grammar1:                 // (top-down, bottom-up)
    @inherit width           // final width
    @inherit th              // temp height, for %s or 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 ? // wordwrap
      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 ? // wordwrap
      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 some values hardcoded
  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
```

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

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

Figure 8: BSS0 passes, constraints, helpers, grammar, and types.

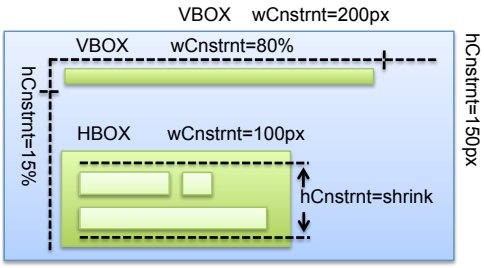



Figure 9: Sample BSS0 layout constraints.

suffices for BSS0 (steps 2 and 4 of Figure 5).

In our larger languages, [14] 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 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 amount 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 opportunties 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, or explicitly or possibly by accident. In contrast, our specification is well-defined.

## 5.3 Parallelization

Attribute grammars expose opportunties for parallelization [2]. First, consider inherited attributes. Data dependencies flow down the tree: given the inherited attributes of

```
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 10: BSS0 parallelization psuedocode. Layout calculations are implemented separately from the scheduling and synchronization traversal function.

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).

A variety of parallel algorithms are now possible. For example, synthesized attributes might be computed with prefix scan operations. While such specialized and tuned operators may support layout subsets, we found much of the layout time in Safari to be spent in general or random-access operations (e.g., $isSVG()$), so we want a more general structure. We take a task-parallel approach (Figure 10). For each node type and grammar, we define custom 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* [10]). One grammar is fully processed before the next. To process a grammar, 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 8 given the attributes in the previous stages; they do not handle concerns like scheduling or synchronization.

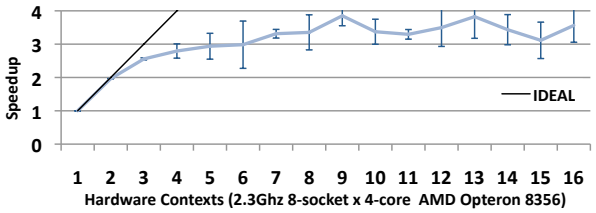
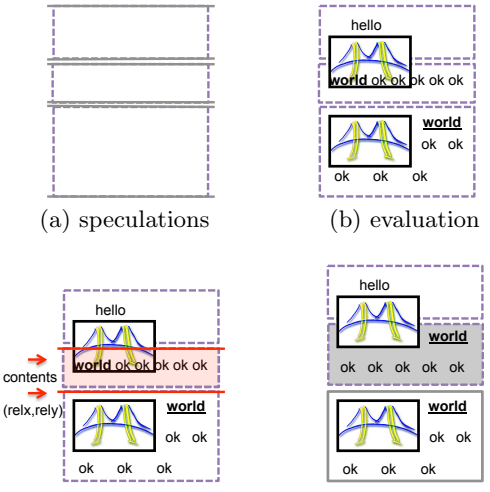



Figure 11: Simulated layout parallelization speedup.

## 5.4 Performance Evaluation

We represented a snapshot of `slashdot.org` using our system and found that box layout time takes only 1-2ms in our simplified model with another 5ms for text layout. In contrast, our profile of Safari reported 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 11 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 overal 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.




(a) speculations (b) evaluation

(c) mispeculation checking (d) reevaluation

Figure 12: Speculative evaluation for floats.

## 5.5 Floats

Advanced layouts such as multiple columns or liquid (multiresolution) flows employ *floating* elements. For example, Figure 12d 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 for more detailed discussion [14].

A revealed weakness of our approach in BSS0 and BSS2 is that floating elements may have long sequential dependencies. For example, notice that attempting to solve the second paragraph in Figure 12b without knowing the positions of elements in the first leads to an incorrect layout. Our solution is to speculatively evaluate grammars (Figures 12a, 12b) and then check for errors (Figure 12c), rerunning grammars that misspeculate (Figure 12d). 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 [17]. 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 – and, beyond the scope of this work, potentially the ability to automatically generate solvers that have them.

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 [14] discusses extending these techniques to richer layout languages. An exemption is that we cannot prove log-time performance for our speculative float algorithm.

## 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 12, 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'.

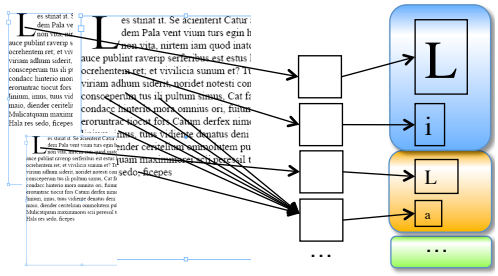



Figure 13: Bulk and parallel font handling.

Figure 13 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 combination of *(character, font face, size,* and *style)* – and then make parallel calls to process this information. We currently perform the pooling step sequentially, but it can be described as a parallel reduction to perform set unions. We use nested `parallel_for` calls, hierarchically encoding affinity on font file and creating tasks at the granularity of (*font, size*).

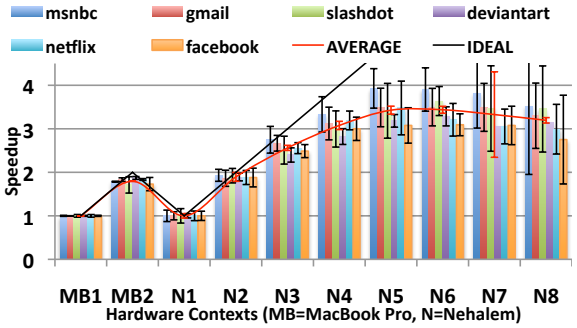



Figure 14: Glyph rendering parallelization speedup.

Figure 14 shows the performance of our algorithm on several popular sites. We use the FreeType2 font library[18], 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 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 from one another, reducing resource contention and hardening against attacks [19]. This leads to pipeline parallelism between components but not for the bulk of time spent within functionally isolated and dependent algorithms.

**Selectors.** Our rule matching algorithm incorporates two sequential optimizations from WebKit. Inspired by our work, Haghighat et al [7] speculatively parallelize matching one selector against one node – the innermost loop of algorithm implicitly within function `match` – but do not show scaling beyond 2 cores nor significant gains on typical sites.

Bordawekar et al [3] study matching XPath expressions against XML files. 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 for many queries and small files. We perform data partitioning, though, in

contrast, we also tile queries. We perform query partitioning by splitting on disjunctions, though this represents a work-inefficient strategy and mostly exists to further empower our redundancy elimination optimization: it is more analogous to static query optimizations. Overall, we find it important to focus on memory, performing explicit reductions, memory preallocation, tokenization, etc. Finally, as CSS is more constrained than XPATH, we perform additional optimizations like right-to-left matching.

**Glyph rendering.** Parallel batch font rendering can already be found in existing systems, though it is unclear how. We appear to be the first to propose tightly integrating it into a structured layout system.

**Specifying layout.** Most document layout systems, like TEX, are implementation-driven, occasionally with informal specifications as is the case with CSS [11, 5]. For performance concerns, they are typically implemented in low-level languages, but, increasingly, high-level languages like Java or ActionScript are used. These are still general purpose; we use an analyzable domain specific language.

**Constraint-based layout.** Executable *specifications* of layout is an open problem. The Cassowary project [4] proposes extending CSS with its linear constraint solver. Unfortunately, linear constraints are insufficient for CSS. A solution is to make a pipeline of linear and ad-hoc solvers [12]. These approaches do not currently support reasoning about layouts, do not have performance on-par with browsers, and elide popular powerful features like floats. In contrast, for a difficult and representative set of rich CSS-like features, we have demonstrated advances in reasoning and performance while still supporting equational reasoning.

**Attribute grammars.** Attribute grammars are a well-studied model [10]. They have primarily been examined as a language for the declarative specification of interpreters, compilers, and analyses for Pascal-like languages. It is not obvious that attribute grammar primitives are appropriate for specifying and optimizing layout. For example, we found multiple passes of parallel (work-stealing) tree traversals to be a suitable parallelization structure, but the only demonstrated support for parallelism in attribute grammars is for decomposing based on regions of the tree. [2] As another concern, we are not aware of any attribute grammar system that meets our need for speculative evaluation.

# 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 definition of layout solving as a series of 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. Furthermore, our approach is amenable to machine-checking and we are even examining automatically generating solvers. This simplifies tasks for browser developers and web designers dependent upon them.

Overall, this work is a milestone in our construction of a parallel, mobile browser for browsing the web on 1 Watt.

# 9. REFERENCES


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

[2] H.-J. Boehm and W. Zwaenepoel. Parallel Attribute Grammar Evaluation. Technical Report TR87-55, Rice University, 1987.

[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] A. Borning, K. Marriott, P. Stuckey, and Y. Xiao. Solving Linear Arithmetic Constraints for User Interface Applications. In *UIST '97: Proceedings of the 10th annual ACM symposium on User interface software and technology*, pages 87–96, New York, NY, USA, 1997. ACM.

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

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

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

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

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

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

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

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

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

[14] L. Meyerovich. A parallel web browser. `http://www.eecs.berkeley.edu/~lmeyerov/projects/pbrowser/`.

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

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

[17] 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.

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

[19] 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.