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 [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, 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 [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 low-level languages: mobile devices suffer 1-2 magnitudes of sequential performance degredation 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 second 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 [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 hope enable developers to instead focus more on application domain concerns.

Webpage processing is a significant bottleneck. Figure 1 compares loadtimes 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: loadtime is still 9x slower

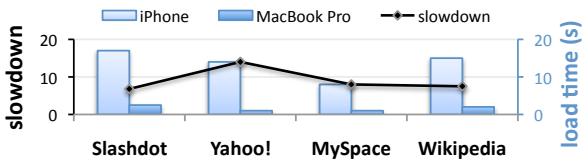


Figure 1: 400Mhz iPhone vs. 2.4Ghz MacBook Pro loadtimes using the same wireless network.

*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 wifi, 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 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 that at least 40% of the time in Safari is spent in these tasks and others report 70% of the time in Internet Explorer [6]. Our paper contributes algorithms for the following presentation tasks in CSS (Cascading Style Sheets [5, 11]):

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. 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 outpout of layout solving is the sizes and positions of elements. CSS layout is a flow-based layout language, which is common to document systems. We developed 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 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.

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 interested in source code, test cases, and benchmarks to our project page [**?**]. This report is an extension of a shorter paper [**?**], including extra benchmarks and, in Section **??**, the specification, algorithms, and proofs for a much larger subset of CSS-like features (we are also working on a translation, but that is well beyond the scope of this work).

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.

We show the basic data flow within a browser in Figure 2 [8]. Loading an HTML page sets off a cascade of events:

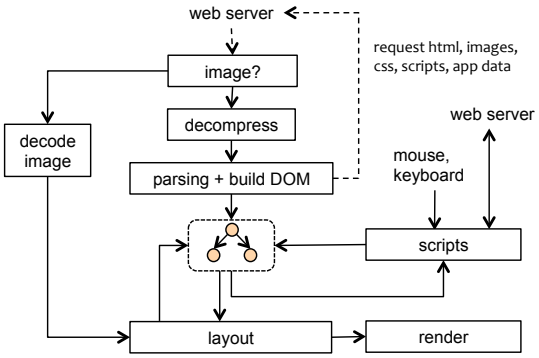


Figure 2: Data flow in a browser.

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 solvend 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.

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 callstack every

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
AVERAGE ms	495	280	233	194	176	113	72	19
AVERAGE %	31	18	15	12	11	7	4	1

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

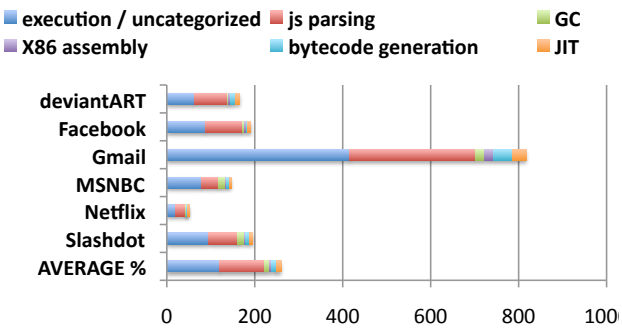


Figure 4: JavaScript task times (ms), 2.4GHz MacBook Pro.

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

20μs, we estimate lowerbounds 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 callstack sampling approach, we ignore 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 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.

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. Surprisingly, more time is spent on parsing related tasks for JavaScript rather than actually running it (Figure **??**). 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 analaogous to our layout one

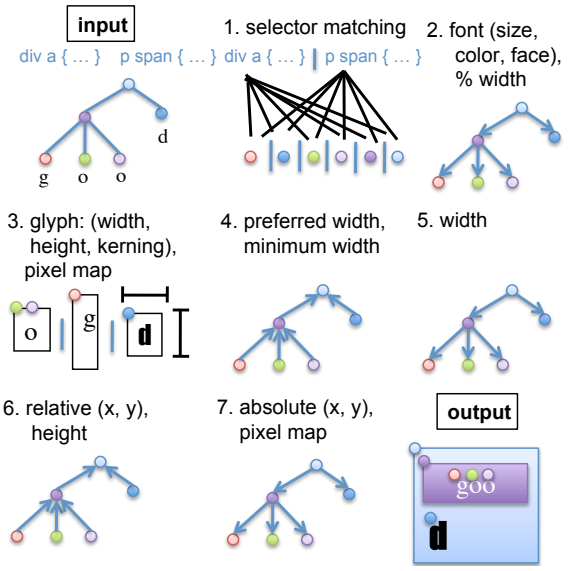


Figure 6: Parallel CSS processing steps.

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 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 lay-

out 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 content, 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:

The translation collapses the terminals `tag`, `id`, and `class` into just `symbol`. For example, consider the HTML node `<div id="account" class="first,on"/>`. Every node has a tag type (e.g., `div`), might have an id (e.g., `account`), and may belong to any number of classes (e.g., `first` and `on`). For the mapping, these four attributes are mapped to one symbol. Figure ?? shows the translation from the common CSS selector language subset to REGEX. 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.

(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
```

$$predicate \text{ ``,'' } predicate \rightarrow \$1 \text{ ``|'' } \$2$$
$$selector \text{ `` < '' } selector \rightarrow \$1 \ \$2$$
$$selector \text{ `` '' } selector \rightarrow \$1 \text{ ``(.*)'' } \$2$$
$$nodePredicate \rightarrow symbol(\$1)$$

Figure 7: Note the equivalence between *nodePredicate* and a REGEX. The predicate corresponds to a REGEX character, except the REGEX character specifies a subset of attributes a node must contain (which is still an equivalence relation) instead of the usual notion of equality.

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.

[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 8: Most of our selector matching algorithm kernel.

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

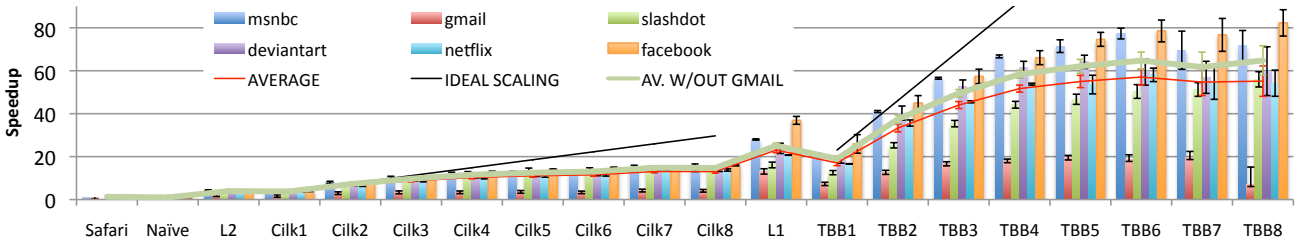


Figure 9: Selector speedup relative to a reimplementation (column 2) of Safari's algorithm (column 1). Labels Cilk<i> and TBB<i> represent the number of contexts. Column Safari on a 2.4GHz laptop, rest on a 4-core × 8-socket 2.3GHz Opteron.

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.

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 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 [17].

In this section, we present our basic specification technique (Section 5.1) for a toy language, some examples of complexities for even this reduced language (Section 5.2), the opportunity for parallelization that it reveals (Section 5.3), a performance experiment (Section 5.4), and some surprising complexity bounds (Section 5.6). For clarity, we use an ongoing example of a toy style sheet language (BSS0). It is not inherently obvious that our technique for BSS0 scales to the common features of CSS that challenge other techniques like linear constraint solving [2]: the next section (Section **??**) shows how to extend these results to reformulate, specify, and parallelize challenging features of CSS like automatic shrink-to-fit sizing and floating elements.

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 Section **??** 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

```
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 10: BSS0 passes, constraints, helpers, grammar, and types.

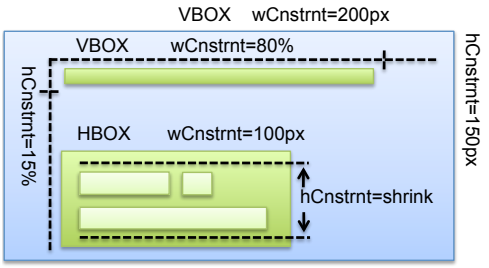


Figure 11: Sample BSS0 layout constraints.

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

In our larger languages, Section **??** 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.

```
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 psuedocode. Layout calculations are implemented separately from the scheduling and synchronization traversal function.

5.3 Parallelization

Attribute grammars expose opportunties for parallelization [3]. 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).

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.

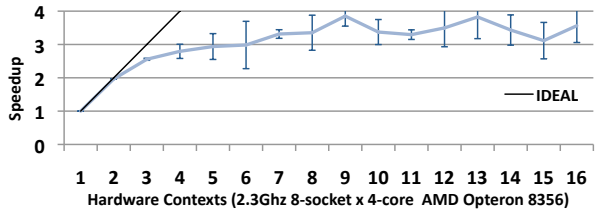


Figure 13: Simulated layout parallelization speedup.

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.

5.5 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.

Section **??** examines extending these results to richer layout language levels.

6. RICHER LAYOUT LANGUAGES

A grammar in the style of BSS0 is too weak to support many CSS features. We examine several common but representatively challenging features of CSS to show how to scale our approach to a more flexible and realistic layout language.

First (Section **??**), we introduce richer shrink-to-fit width constraints in BSS1, motivating the use of a pipeline of grammars rathers than just one. Parallelization is analogous to that of BSS0. We then overview extensions for text and absolute and fixed positioning, finding them straightforward in our framework. Next, we introduce BSS2 (Section **??**), which supports a more perplexing feature: floating elements. Floating elements break our parallelization strategy, but they are foundations for modern layouts so we should support them: we show how to adapt our parallel algorithm to be *speculative* in the otherwise linearizing scenarios (Section **??**). Finally, we revisit analytic reasoning about termination and complexity guarantees for these richer languages (Section **??**).

6.1 BSS1: Shrink-to-fit Constraints

BSS1 is like BSS0 except it supports shrink-to-fit for widths (depicted in Figure **??**). Our first concern is how to *correctly* solve layout constraints. A grammar in the style of BSS0 is too weak for implementing BSS1. The CSS specification is written informally and guidelines for individual features are spread throughout the standard – or are not even defined. Consider setting `wCnstrnt=shrink` and solving for the width:

$$width = \min(\max(minWidth, availWidth), prefWidth)$$

The intuition is that the width should be big enough to contain its children without having them overflow ($minWidth$) and would look best with a width of $prefWidth$, but must not be bigger than the parent node's width ($availWidth$). Different boxes, such as those that position their children vertically or horizontally, calculate $minWidth$ and $prefWidth$ differently... in an undefined way. In the case of a box vertically positioning its children, both $minWidth$ and $prefWidth$

```
interface Node
  @input children, prev, wCnstrnt, hCnstrnt
  @input defCnstrnt       // for when parent.tw is NaN
  @grammar1:              // (top-down, bottom-up)
    @inherit tw, th       // partial width/height solution
    @synthesize minWidth  // ex: widest word in a paragraph
    @synthesize prefWidth // ex: sum of words in a paragraph
  @grammar2:              // (top-down, bottom-up)
    @inherit width        // final width
    @synthesize height    // final height
    @synthesize relx, rely// position relative to parent
  @grammar3:              // (top-down)
    @inherit x, y         // final, absolute position

class V implements Node
  @grammar1.inherit       // top-down
  for c in children:
    c.th = size(th, c.hCnstrnt, auto)
    c.tw = size(tw, c.wCnstrnt, defCnstrnt)

  @grammar1.synthesize    // bottom-up
  prefWidth = join(tw, max([c.prefWidth | c in children]))
  minWidth = join(tw, max([c.minWidth | c in children]))

  @grammar2.inherit       // top-down
  for c in children:
    c.width =  w(width, c.tw, c.minWidth, c.prefWidth)

  @grammar2.synthesize    //bottom-up
  children[0]rely = 0
  for c > 0 in children:  // note loop-carried dependence
    c.rely = c.prev.rely + c.prev.height
  for c in children:
    c.relx = 0
  height = join(th, sum([c.height | c in children]))

  @grammar3.inherit       // top-down
  for c in children:
    c.x = x + c.relx
    c.y = y + c.rely

class H implements Node
  @grammar1.inherit
  for c in children:
    c.th = size(th, c.hCnstrnt, auto)
    c.tw = size(tw, c.wCnstrnt, defCnstrnt);

  @grammar1.synthesize
  prefWidth = join(tw, sum([c.prefWidth | c in children]))
  minWidth  = join(tw, max([c.minWidth | c in children]))

  @grammar2.inherit
  for c in children:
    c.width = w(width, c.tw, c.minWidth, c.prefWidth)

  @grammar2.synthesize
  let (_, _, highest) =
    foldl(
      function (xc, yc, hc) c =>
        let overflow = xc > 0 && (xc + c.width) > width in
        c.relx = overflow ? 0 : xc
        c.rely = overflow ? yc + hc : yc
        return (c.relx + c.width, c.rely,
                overflow ? c.height : max(hc, c.height))
      (0, 0, 0), children) in
  height = join(th, highest)

  @grammar3.inherit
  for c in children:
    c.x = x + c.relx
    c.y = y + c.rely

class Root constrains V   // V node with some values hardcoded
  th = 100, tw = 100      // browser specifies all of these
  minWidth = 100, prefWidth = 100
  width = 100, height = 100
  relx = 0, rely = 0
  x = 0, y = 0
```

Figure 14: BSS1 passes and constraints.

```
function size (auto | expand | shrink, p %, d) -> d
           | (v px, p %, d) -> v * 0.01 * p px
           | (v, p px, d) -> p px
           | (v, pol, d) -> pol
function join (auto | expand | shrink, v) -> v
           | (p px, v) -> p
function w (a, expand | auto, m, p) -> a
         | (a, shrink, m, p) -> min(max(m, a), p)
         | (a, v px, m, p) -> v
```

$$R \rightarrow V \mid H$$
$$V \rightarrow H^* \mid V^*$$
$$H \rightarrow V^*$$

$$\begin{aligned}
V ::& \{\text{wCnstrnt} : DIM_w, \text{hCnstrnt} : DIM_h, \text{defCnstrnt} : POL_w, \\
& \text{children} : V\ list,\ \text{prev} : V, \\
& \text{tw} : P \mid POL_w,\ \text{th} : P \mid \text{auto},\ \text{minWidth} : P,\ \text{prefWidth} : P, \\
& \text{width} = P,\ \text{relx} : P,\ \text{rely} : P,\ \text{height} : P,\ \text{x} : P,\ \text{y} : P\ \} \\
\textit{Root} ::& V \text{ where } \{\text{width} : P,\ \text{height} : P,\ \text{tw} : P,\ \text{th} : P,\ \text{x} : P,\ \text{y} : P\ \} \\
DIM_w ::& P \mid PCNT \mid POL_w \\
DIM_h ::& P \mid PCNT \mid \text{auto} \\
P ::& \mathbb{R}\ \text{px} \\
PCNT ::& \mathbb{P}\ \%\ \text{where } \mathbb{P} = [0,1] \subset \mathbb{R} \\
POL_w ::& \text{expand} \mid \text{shrink}
\end{aligned}$$

Figure 15: BSS1 helpers and types.

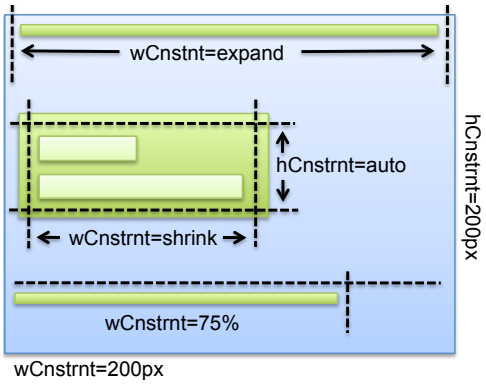


Figure 16: Sample BSS1 layout constraints.

would be the widest child. In a paragraph of text, the $prefWidth$ would be the sum of word lengths.

It is not obvious how to solve a layout that contains shrink-to-fit formulas. The width of a node is typically synonomous with $availWidth$, so a `shrink` node depends on its parent's width. Furthermore, it is a function of the preferred and minimum widths of its children: is there a cyclic dependency on widths between parent and children nodes? Assume we know how to solve such dependencies, such as a traversal pattern over the HTML tree as browsers typically do: is this traversal structure correct for other featurs? In our initial implementation, based just on reading the CSS specification, we had multiple false starts.

Our solution is to solve layouts using a *sequence* of our restricted attribute grammars. Consider solving for shrink-to-fit widths. Just as we have intermediate attribute `th` in BSS0, we introduce `tw`, its analogue for widths. We then define $minWidth$ and $prefWidth$ attributes in terms of ones below them and `tw`, so, assuming a previous inherited pass for `tw` values, we can solve them all in a subsequent upwards synthesis pass. By creating these first two passes, we can now solve for the width in a final downwards inherited attribute passing using the above min/max equation. Figure 5 visualizes these inherited (downwards) and synthesized (upwards) attribute passes for BSS1. The first set of definitions in Figure **??** specify the sequence of grammars (@someGrammar:prevGrammar) and annotates computed attributes as either inherited (@inherit) or synthesized (@synthesize): it describes how to implement BSS1 as a sequence of tree traversals.

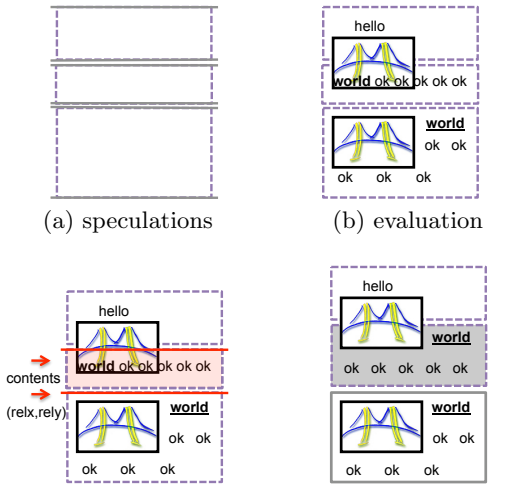
BSS1 has several subtleties. First, we introduce definitions not present within CSS: as noted, CSS does not define $minWidth$ etc. We believe our definition is intuitive and similar to what browsers implement, and as will be clear, enables parallelization. Second, as in BSS0, our language subset contains CSS constraints with suprising solutions. Analogous to the problem of `shrink` height constraints in BSS0 (Section 5.2), consider the following subtree:

V[wCnstrnt=shrink](V[wCnstrnt=50%](V[wCnstrnt=20px]))

As before, in a fixedpoint solution, the two outer nodes would have a solved width of 0px. Browsers instead pick 20px. The CSS specification does not define what to do; we pick the browser approach as it seems more intuitive, is simple to parallelize, and do not introduce iterative fixedpoint computations. As a more flexible approach than we took in BSS0, we introduce attribute $defCnstrnt$ to pick a layout policy for when percentage constraint like the above are discarded. Finally, analogous to BSS0, width and height solutions of parent nodes do not imply any sort of visual containment relationship for their children.

6.2 Specifying Floats

We introduce CSS's *float* attribute in BSS2. Figure 12d depicts a layout where the image in the first paragraph has the attribute `float=left`. Its containing box may end above the bottom of the image, and sibling elements of the containing box must position themselves around it. A float is on the left or right edge of its containing box (unless it would hit another float). Complicating reasoning, this may displace preceeding elements: seemingly, positioning requires multiple passes forcing questions of reflows and even nontermination. We found all browsers to interpret the float attribute differently, which is problematic because modern layouts (e.g.,



(a) speculations (b) evaluation

(c) mispeculation checking (d) reevaluation

Figure 17: Speculative evaluation for floats.

multi-column ones) typically depend upon floats for their overall layout structure.

Nodes interact with floating elements in a variety of ways. We refactored CSS to enable control over three orthogonal interactions (not all of which are exposed by CSS):

- **Floating.** Attribute f controls whether an element is floating or should be positioned normally. We currently only support floating to the left; floating to the right seems analogous.
- **Escaping floats.** Using attribute e, an element may specify whether or not to expose floats within it to elements outside of it.
- **Including floats.** Floating elements descending from an element may extend further than normal elements. Attribute i controls whether these elements should be considered in measuring the boundary.

Specifying floats is not obvious. We describe the intuition for our specification here and defer discussion of algorithmic details for the remainder of this section. Floating elements earlier in the tree may impact the positioning of all subsequent elements. We therefore thread floats throughout the grammar: floats before a node in the tree are stored in attribute *spaceIn* and floats before or below a node in the tree are stored in *spaceOut*. As an intuition, the *spaceIn* of a node is based on the *spaceOut* of its sibling and the *spaceIn* of the first sibling is based on the *spaceIn* of its parent.

A peculiarity of CSS is that such information does not need to be given to children of an H node. A further subtlety in horizontal nodes is that the position of any child is not known until all neighbors in the same line are placed due to concerns like unknown heights and floats following normal elements (displacing the normal one if it was placed if there were no subsequent floating elements). Our grammar therefore build up a list of elements that fit on a line, arranges them, and then proceeds.

We overview various variables and their invariants in our algorithm. We do not store all of them as explicit attributes. Our attribute grammar reifies the children of a node as a list, so we use fold expressions with explicit accumulator variables. These would become synthesized attributes if we were to rewrite our grammar to use a list production rather than V^* and H^*. Insight to their meaning and use is as follows:

- *cursor.* The next position where a non-floating element might be placed.
- *space.* Preceeding floating elements (represented by the bottom right coordinate) around which subsequent floats must be placed.
- *space.fringe.* The absolute coordinates of preceeding floats that an element must be placed around. An invariant maintained when adding an element (with `UPlus`) is that elements (stored right-to-left) have increasing y coordinates.
- *space.shift.* The relative offset of the node. We keep track of this because node calculations are relative but floats are reused and therefore global; knowing it enables us to convert between absolute and global coordinates. Functions `shift` (`unshift`) move the relative coordinate system along (against) a vector.
- *space.by, rx, and ty.* We keep track of the bottom of the bottom-most float, right of the bottom-most float, and top of the last positioned (non-*fqueue*) float, respectively. These coordinates are relative to the node whose children are being positioned.
- *nQueue, fQueue.* H nodes need to keep track of more information than V nodes. In particular, these values are a list of unplaced non-floating (normal) and floating elements, respectively. The normal elements can all fit on the same line; when a normal element is encountered that requires word-wrapping, function `clear` may be used to place all of them. Floating elements in *fQueue* are those that go on *subsequent* lines: they cannot be placed until the current line's height is known, which is unknown until all elements that are to be placed upon it are known... and subsequent non-floating elements may actually go on a preceeding line. In contrast, floats on the current line can be placed immediately.
- *lh.* Line height (of non-float elements). In our current formulation, only H boxes have lines, though a possible interpretation for V boxes is to have each non-floating element be considered line.

6.3 Speculative Execution to Parallelize Floats

Our specification of floats uses attributes *spaceIn* and *spaceOut* to represent elements that may impact layout. These attributes reveal long dependencies between otherwise disjoint HTML subtrees during phase `@grammar2.synthesize`, challenging parallelization. Consider the bridge in the first paragraph of Figure 12d: we cannot position the text in the second paragraph until we know how far the bridge enters into it.

Usefully, floats are rare. E.g., used only for columns, there will only be a few floats for potentially thousands of other nodes. Figure 12 depicts a speculative algorithm where we lay out each subtree under the assumption that other subtrees do not have floats. A computed attribute for nodes is

whether floats nested within them escape out to potentially impact other subtrees: we only need to recompute subtrees if they follow ones with escaping floats. Furthermore, the amount to be recomputed may be limited (e.g., in third paragraph, we only need to reposition the outer box, not recompute its contents). Given our current need to simulate layout slowdowns, we want to further build up our layout engine before examining float performance.

In terms of our grammar, our speculation for a node is that attribute *spaceIn* is `mtSpace` and, when considering a child node's attributes, `spaceOut` is the same as `spaceIn`. Thus, in both `@grammar2.synthesize` rule sections, when `spaceIn` and `c.spaceOut` are accessed, a speculative value is always available. Furthermore, even during a mispeculation for a particular node, most of the node's computations might not be impacted: floats that are not near it may be moved.

6.4 BSS1 and BSS2 Termination and Complexity

Formally, BSS1 is like BSS0 (Section 5.6) except, instead of using one grammar, it uses a sequence of 3 grammars that are structurally similar to BSS0 grammars. This presentation is intentional: traditional attribute grammars would combine the full specification into one grammar, but would not guarantee distinct single-pass inherited and then synthesized attribute solving phases. In our presentation, the number of passes is a constant factor; as with BSS0, layout is linear in the amount of tree nodes, and, for a balanced tree, $O(log|tree|)$ time with parallelization. The passes and their parallelization are depicted in Figure 5.

The extended language BSS2, which uses floats, is more complicated. Generalizing to singly-assigned variables, our proofs holds for the sequential case: in any place an iteration occurs, it is over nodes, and at most once. The subtlety is in managing lists of unplaced floats and normal elements, being careful to only traverse an element not in head position at most once. These properties can be manually verified; layout solving is $O(|tree|)$ and reflow is unnecessary (instead positioning is delayed using the `queue` accumulator). Note that the second stage of BSS2, consisting of a downwards and then upwards pass in the speculative case, should be reformulated as a single recursive descent for this scenario.

For the parallel case, if our speculations are correct (e.g., if there are no floats), worst-case time is again $O(log|tree|)$. It is unclear how to reevaluate the grammar if there are mispeculations. For example, the sequential version may be switched in, guaranteeing worst-case linear time. However, we expect some mispeculations, but only in a limited area. An alternative is to perform the downwards and the upwards passes until a fixedpoint is reached, but this has worst-case time $O((log|tree|)|tree|)$ as at most one directed edge is solved per iteration. This worst-case is unlikely; we are working on providing a clearer bound relative to the amount of mispeculations (which is linked to the second $|tree|$ term above, signifying iterations to correct and propagate values).

7. 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

```
interface Node
  @input children, prev, wCnstrnt, hCnstrnt
  @input defCnstrnt       // for when parent.tw is NaN
  @input e, f, i          // float controls
  @speculate (spaceIn, spaceOut) = (mtSpace, spaceIn)
  @grammar1:              // (top-down, bottom-up)
    @inherit tw, th       // partial width/height
    @synthesize minWidth  // ex: widest word in a phrase
    @synthesize prefWidth // ex: sum of words in a phrase
  @grammar2:              // (top-down, bottom-up)
    @inherit width        // final width
    @synthesize spaceIn   // floats from neighbors
    @synthesize spaceOut  // exposed floats from siblings
    @synthesize relx, rely// position relative to parent
    @synthesize height    // final height
  @grammar3:              // (top-down)
    @inherit x, y         // final, absolute position

class V implements Node
  @grammar1.inherit       // top-down
  for c in children:
    c.th = size(th, c.hCnstrnt, auto)
    c.tw = size(tw, c.wCnstrnt, defCnstrnt)

  @grammar1.synthesize    // bottom-up
  prefWidth = join(tw, max([c.prefWidth
        | c in children and (i or not c.f)]))
  minWidth = join(tw, max([c.minWidth
        | c in children and (i or not c.f)]))

  @grammar2.inherit
  for c in children:
    c.width = w(width, c.tw, c.minWidth, c.prefWidth)

  @grammar2.synthesize
  (cursor, space) =
    foldl(
      function (cursorAcc, spaceAcc) c =>
        if c.f:
          c.spaceIn = mtSpace
          (c.relx, c.rely) =
            delta_f(cursorAcc, width, spaceAcc,
                    c.width, c.height, 0)
          return (cursorAcc,
                  UPlus(spaceAcc,
                        {x: c.relx, w: c.width,
                         y: c.rely, h: c.height}))
        else:
          c.relx = 0
          c.rely = cursorAcc.y
          c.spaceIn = shift(spaceAcc, cursorAcc)
          return ({x: 0, y: cursorAcc.y + c.height},
                  unshift(c.spaceOut, cursorAcc)),
      ({x: 0, y: 0}, f ? spaceIn : mtSpace),
      children)
  spaceOut = e ? space : spaceIn
  height = join(th, i ? max(cursor.y, space.by)
                      : cursor.y)

  @grammar3.inherit
  for c in children:
    c.x = x + c.relx
    c.y = y + c.rely
```

```
class H implements Node
  @grammar1.inherit
  for c in children:
    c.th = size(th, c.hCnstrnt, auto)
    c.tw = size(tw, c.wCnstrnt, defCnstrnt);

  @grammar1.synthesize
  prefWidth = join(tw, sum([c.prefWidth
        | c in children and (i or not c.f)]))
  minWidth = join(tw, max([c.minWidth
        | c in children and (i or not c.f)]))

  @grammar2.inherit
  for c in children:
    c.width = w(width, c.tw, c.minWidth, c.prefWidth)

  @grammar2.synthesize
  let (cursor, mostSpace) =
    foldl(
      function (cursorAcc, spaceAcc) c =>
        c.spaceIn = mtSpace
        if c.f:
          (fx, fy) =
            delta_f(cursorAcc, width, spaceAcc.space,
                    c.width, c.height, spaceAcc.lh)
          if fy = cursorAcc.y:
            (c.relx, c.rely) = (fx, fy)
            return
              ({x: cursorAcc.x + c.width, y: cursorAcc.y},
               {space:
                  UPlus(spaceAcc.space,
                    {x: c.relx, w: c.width,
                     y: c.rely, h: c.height}),
                nQueue: spaceAcc.nQueue, fQueue: [],
                lh: spaceAcc.lh})
          else:
            return
              (cursorAcc,
               {space: spaceAcc, nQueue: spaceAcc.nQueue,
                fQueue: [c] @ spaceAcc.fQueue,
                lh: spaceAcc.lh})
        else:
          let cursorAcc' =
            place(spaceAcc, cursorAcc, width, c.width) in
          return
            (cursorAcc',
             cursor.y = cursorAcc'.y ?
                 {space: spaceAcc.space,
                  nQueue: [c] @ spaceAcc.nQueue,
                  fQueue = spaceAcc.fQueue,
                  lh: max(spaceAcc.lh, c.height)}
               : {space: clear(spaceAcc, width, cursorAcc),
                  nQueue: [c], fQueue[],
                  lh: c.height})
      ({x: 0, y: 0},
       {space: spaceIn, nQueue: [], fQueue: []}),
      children),
    space = {space: clear(mostSpace, width, cursor),
             nQueue: [], fQueue: []},
    lh = max([i.height | i in mostSpace.nQueue]) in
  spaceOut = e ? space.space : spaceIn
  height = join(th,
            i ? max(cursor.y + lh, space.space.by)
              : cursor.y + lh

  @grammar3.inherit
  for c in children:
    c.x = x + c.relx
    c.y = y + c.rely

class Root constrains V
  th = 100, tw = 100
  minWidth = 100, prefWidth = 100
  width = 100, height = 100
  relx = 0, rely = 0
  x = 0, y = 0
  spaceIn = mtSpace
```

$$R \to V \mid H$$
$$V \to H^* \mid V^*$$
$$H \to V^*$$

$$V ::=_{attribs} [\cdots,\ {}_c\mathrm{f} = F,\ {}_c\mathrm{e} = \mathbb{B},\ {}_c\mathrm{i} = \mathbb{B}]$$
$$H ::=_{attribs} [\cdots,\ {}_c\mathrm{f} = F,\ {}_c\mathrm{e} = \mathbb{B},\ {}_c\mathrm{i} = \mathbb{B}]$$
$$\cdots$$
$$F ::= \text{none} \mid \text{left}$$
$$B ::= \text{true} \mid \text{false}$$

Restriction for CSS: a child of an H node has attribute ${}_c\mathrm{e} = \text{false}$.

Figure 18: BSS2 input attributes subsume BSS1's.

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 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 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.

8. 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 [4] study matching XPath expressions against XML files. They experiment with *data partitioning*

```
mtSpace =
  {fringe: [], shift: {x: 0, y: 0}, by: 0, rx: 0, ty: 0}

UPlus
  {fringe: f, by: by, rx: rx, ty: ty,
   shift: {x: sx, y: sy}}
  {x: x, w: w, y: y, h: h}
  = match f with
    | []
      => {fringe: [(sx + x + w, sy + y + h)],
          shift: {x: sx, y: sy},
          by: max(by, y + h),
          rx: max(rx, x + w),
          ty: y}
    | (hx, hy) :: tl
      => hy > sy + y + h ?
          {fringe: [(sx + x + w, sy + y + h),
                    (hx, hy)] @ tl,
           shift: {x: sx, y: sy},
           by: max(by, y + h),
           rx: max(rx, x + w),
           ty: y}
        : UPlus({fringe: tl, by: by, rx: rx, ty: ty,
                 shift: {x: sx, y: sy}},
                {x: x, w: w, y: y, h: h})

shift {fringe: f, shift: {x: sx, y: sy},
       by: by, rx: rx, ty: ty}
      {x: x, y: y}
  = {fringe: f, shift: {x: sx + x, y: sy + y},
     by: by - y, rx: rx - x, ty: ty - y}

unshift space {x: x, y: y} = shift(space, {x: -x, y: -y})

place {space: _, nQueue: nonfloats, fQueue: _}
      {x: x, y: y} nw cw
 = x + cw > nw ?
     {x: cw, y: y + max([i.height | i in nonfloats])}
   : {x: x + cw, y: y}

delta_f
  {x: cx, y: cy} w
  {fringe: f, by: by, rx: rx, ty: ty,
   shift: {x: sx, y: sy}}
  wf hf lh
  = match f with
    | [] => (0,
        max(by, cx > 0 && cx + wf > w ?
                  cy + lh : cy)
    | (hx, hy) :: tl
      => let t = max(cy, ty) in
        t < hy - sy ?
            hx - sx + wf < w ?
                (cx, t)
              : delta_f(
                  {x: cx, y: hy - sy}, w,
                  {fringe: tl, by: by, rx: rx, ty: ty,
                   shift: {x: sx, y: sy}},
                  wf, hf, lh)
          : delta_f(
              {x: cx, y: cy}, w,
              {fringe: tl, by: by, rx: rx, ty: ty,
               shift: {x: sx, y: sy}},
              wf, hf, lh)

clear {space: space, nQueue: nonfloats, fQueue: floats}
      w {x: x, y: y} =
  let lh = max([i.height | i in nonfloats]),
      ix = foldl(
        function ix i =>
          (i.relx, i.rely) = (ix, y + lh - i.height)
          ix + i.width,
        x - sum([i.width | i in nonfloats]),
        nonfloats) in
  foldl(
    function spaceAcc f =>
      (f.x, f.y) =
        delta_f({x: x, y: y}, w, spaceAcc, f.w, f.h, lh)
        UPlus(spaceAcc,
              {x: f.x, w: f.width, y: f.y, h: f.height}),
    space,
    floats)
```

Figure 19: BSS2 layout computation phase 2 helpers.

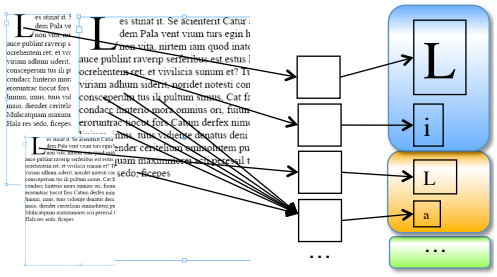


Figure 20: Bulk and parallel font handling.

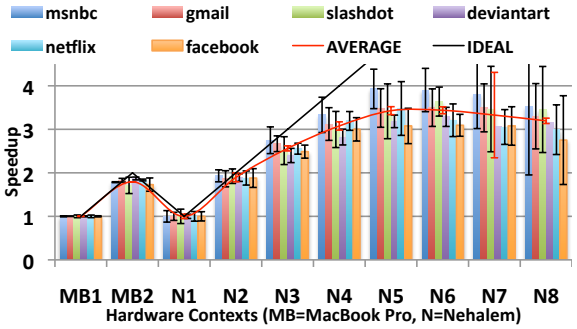


Figure 21: Glyph rendering parallelization speedup. Column label shows number of hardware contexts used.

(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 [2] proposes extending CSS with its linear constraint solver. While prioritized linear constraints can model inheritance, they cannot express the box model. 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.

Manual layout optimizations. Brown proposes parallelizing layout [**?**] for document languages where points of independence are known, such as page boundaries: these are typically unknown in rich layout languages and we have found no reports of being able to parallelize document layout. MacDonald et al propose speculative partial evaluations for layout [**?**] and support a layout language that does so. The language is simple – closer to BSS0 than BSS1 – and it is not clear that the specification is structured in an extensible way. Furthermore, the speculation is for partial evaluation, where multiple variants of a dynamic value are predicted: multiple partial layouts are computed based on them. While partial evaluation may be applicable to optimizations like incremental layout, it is not clear how to apply them to breaking dependencies as we require for parallel processing of layouts or where to use multiple variants while avoiding a decrease in work and time efficiency. In contrast, we show where to speculatively break sequential dependencies to enable parallel computation.

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. [3] As another concern, we are not aware of any attribute grammar system that meets our need for speculative evaluation.

Painting. Painting solved layouts may be hardware accelerated. Glitz [**?**] reroutes window manager shape painting commands to GPUs, but does not have reliable speedups. It uses *immediate* rendering, where essentially blocking commands to paint shapes are transmitted: we believe an algorithm informed by the layout stage is a promising alternative as batches of independent commands may be sent. Finally, specialized wide instructions like SSE might be used in kernels like painting vectors such as boxes and glyphs. Our approach exposes higher levels of parallelism; the degrees of parallelism introduced by the approaches might be multipled for total gains.

9. 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.

10. SUPPORT

Krste Asanovic, Chan Siu Man, Chan Siu On, Chris Jones, and Robert O'Callahan provided valuable help and discussion in early stages of this work. Heidi Pan, Ben Hindman, Rajesh Nishtala, Shoaib Kamil, Andrew Waterman, and other members of the Berkeley Parallelism Lab provided implementation advice.

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

[4] 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.

[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.