Parallelizing the Web Browser

Christopher Grant Jones, Rose Liu, Leo Meyerovich, Krste Asanović, Rastislav Bodík
Department of Computer Science, University of California, Berkeley
{cgjones,rfl,lmeyerov,krste,bodik}@cs.berkeley.edu

Abstract

We argue that the transition from laptops to handheld computers will happen only if we rethink the design of web browsers. Web browsers are an indispensable part of the end-user software stack but they are too inefficient for handhelds. While the laptop reused the software stack of its desktop ancestor, solid-state device trends suggest that today's browser designs will not become sufficiently (1) responsive and (2) energy-efficient. We argue that browser improvements must go beyond JavaScript JIT compilation and discuss how parallelism may help achieve these two goals. Motivated by a future browser-based application, we describe the preliminary design of our parallel browser, its work-efficient parallel algorithms, and an actor-based scripting language.

1 Browsers on Handheld Computers

Bell's Law of Computer Classes predicts that handheld computers will soon replace and exceed laptops. Indeed, internet-enabled phones have already eclipsed laptops in number, and may soon do so in input-output capability as well. The first phone with a pico-projector (Epoq) launched in 2008; wearable and foldable displays are in prototyping phases. Touch screens keyboards and speech recognition have become widely adopted. Continuing Bell's prediction, phone sensors may enable applications not feasible on laptops.

Further evolution of handheld devices is limited mainly by their computing power. Constrained by the battery and heat dissipation, the mobile CPU noticeably impacts the web browser in particular: even with a fast wi-fi connection, the iPhone may take 20 seconds to load the Slashdot home page. Until mobile browsers become dramatically more responsive, they will continue to be used only when a laptop browser is not available. Without a fast browser, handhelds may not be able to support compelling web applications, such as Google Docs, that are sprouting in laptop browsers.

One may expect that mobile browsers are network-bound, but this is not the case. On a 2Mbps network connection, soon expected on internet phones, the CPU bottleneck becomes visible: On a ThinkPad X61 laptop, halving the CPU frequency doubles the Firefox page load times for cached pages; with a cold cache, the load time is still slowed down by 50%. On an equivalent network connection, the iPhone browser is 5 to 10-times slower than Firefox on a fast laptop. The browser is CPU-bound because it is a compiler (for HTML), a page layout engine (for CSS), and an interpreter (for JavaScript); all three tasks are on a user's critical path.

The successively smaller form factors of previous computer generations (workstations, desktops, and then laptops) were enabled by exponential improvements in the performance of single-thread execution. Future CMOS generations are expected to increase the clock frequency only marginally, and handheld application developers have already adapted to the new reality: rather than developing their applications in the browser, as is the case on the laptop, they rely on native frameworks such as the iPhone SDK (Objective C), Android (Java), or Symbian (C++). These frameworks offer higher performance but do so at the cost of portability and programmer productivity.

2 An Efficient Web Browser

We want to redesign browsers in order to improve their (1) responsiveness and (2) energy efficiency. While our primary motivation is the handheld browser, most improvements will benefit laptop browsers equally. There are several ways to achieve the two goals:

- *Offloading the computation.* Used in the Deepfish and Skyfire mobile browsers, a server-side proxy browser renders a page and sends compressed images to a handheld. Offloading bulk computations, such as speech recognition, seems attractive, but adding server latencies exceeds the 40ms threshold for visual perception, making proxy architectures insufficient for interactive GUIs. Disconnected operation is inherently impossible.
- *Removing the abstraction tax.* Browser programmers pay for their increased productivity with an abstraction tax—the overheads of the page layout engine, the JavaScript interpreter, parsers for applications deployed in plain text, and other components. We measured this tax to be two orders of magnitude: a small Google Map JavaScript application runs about 70-times slower than the equivalent written using C and pixel frame buffers [3]. Removing the abstraction tax is attractive because

it improves both responsiveness (the program runs faster) and energy efficiency (the program performs less work). JavaScript overheads can be reduced with JIT compilation, typically 5 to 10-times, but browsers often spend only 15% of time in the interpreter. Abstraction reduction remains attractive but we leave it out of this paper.

- *Parallelizing the browser.* Future CMOS generations will not allow significantly faster clock rates, but they will be about 25% more energy efficient per generation. The savings translate into additional cores, already observed in handhelds. Parallelizing the browser improves responsiveness (goal 1) and to some extent also energy efficiency (goal 2): vector instructions improve energy efficiency per operation, and although parallelization does not decrease the amount of work, gains in program acceleration may allow us to slow down the clock, further improving the energy efficiency.

The rest of this paper discusses how one may go about parallelizing the web browser.

3 What Kind of Parallelism?

The performance of the browser is ultimately limited by energy constraints and these constraints dictate optimal parallelization strategies. Ideally, we want to minimize energy consumption while being sufficiently responsive. This goal motivates the following questions:

1. Amount of parallelism: Do we decompose the browser into 10 or 1000 parallel computations? The answer depends primarily on the parallelism available in handheld processors. In five years, 1W processors are expected to have about four cores. With two threads per core and 8-wide SIMD instructions, devices may thus support about 64-way parallelism.

The second consideration comes from voltage scaling. The energy efficiency of CMOS transistors increases when the frequency and supply voltage are decreased; parallelism accelerates the program to make up for the lower frequency. How much more parallelism can we get? There is a limit to CMOS scaling benefits: reducing clock frequency beyond roughly 3-times from the peak frequency no longer improves efficiency linearly [6]. In the 65nm Intel Tera-scale processor, reducing the frequency from 4Ghz to 1GHz reduces the energy per operation 4-times, while for the ARM9, reducing supply voltage to near-threshold levels and running on 16 cores, rather on one, reduces energy consumption by 5-times [15]. It might be possible to design mobile processors that operate at the peak frequency allowed by the CMOS technology in short bursts (for example, 10ms needed to layout a web page), scaling down the frequency for the remainder of the execution; the scaling may allow them to exploit additional parallelism.

What are the implications of these trends on browser parallelization? Consider the goal of sustained 50-way parallelization execution. A pertinent question is how much of the browser we can afford *not* to parallelize. Assume that the processor can execute 250 parallel operations. Amdahl's Law shows that to to sustain 50-way parallelism, only 4% of the browser execution can remain sequential, with the rest running at 250-way parallelism. Obtaining this level of parallelism is challenging because, with the exception of media decoding, browser algorithms have been optimized for single-threaded execution. This paper suggests that it may be possible to uncover 250-way parallelism in the browser.

2. Type of parallelism: Should we exploit task parallelism, data parallelism, or both? Data parallel architectures such as SIMD are efficient because their instruction delivery, which consumes about 50% of energy on superscalar processors, is amortized among the parallel operations. A vector accelerator has been shown to increase energy efficiency 10-times [9]. In this paper, we show that at least a part of the browser can be implemented in data parallel fashion.

3. Algorithms: Which parallel algorithms improve energy efficiency? Parallel algorithms that accelerate programs do not necessarily improve energy efficiency. The handheld calls for parallel algorithms that are work efficient—i.e., they do not perform more total work than a sequential algorithm. An example of a work-inefficient algorithm is speculative parallelization that misspeculates too often. Work efficiency is a demanding requirement: for some "inherently sequential" problems, such as finite-state machines, only work-inefficient algorithms are known [5]. We show that careful speculation allows work-efficient parallelization of finite state machines in the lexical analysis.

4 The Browser Anatomy

Original web browsers were designed to render hyperlinked documents. Later, JavaScript programs, embedded in the document, provided simple animated menus by dynamically modifying the document. Today, AJAX applications rival their desktop counterparts.

The typical browser architecture is shown in Figure 1. 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 inlined into the DOM. 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.

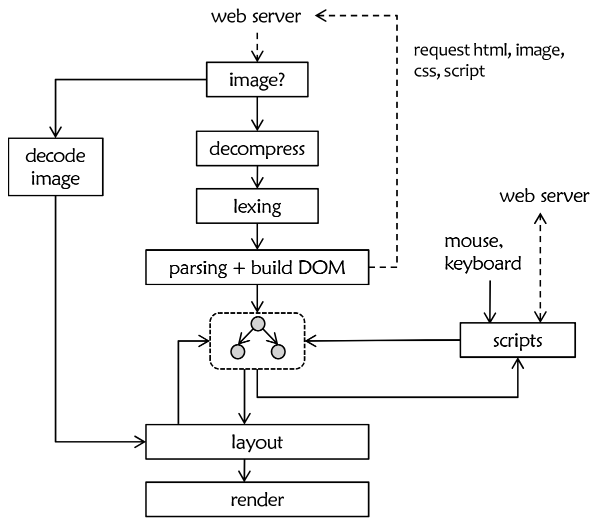

Figure 1: The architecture of today's web browser.

5 Parallelizing the Frontend

The browser frontend compiles an HTML document into its object model, JavaScript programs into bytecode, and CSS style sheets into rules. These parsing tasks take more time that we can afford to execute sequentially: Internet Explorer 8 reportedly spends about 3-10% of its time in parsing [13]; Firefox, according to our measurements, spends up to 40% time in parsing. Task-level parallelization can be achieved by parsing downloaded files in parallel. Unfortunately, a page is usually comprised of a small number of large files, necessitating parallelization of single-file parsing. Pipelining of lexing and parsing may double the parallelism in HTML parsing; the lexical semantics of JavaScript prevents the separation of lexing from parsing.

To parallelize single-file text processing, we have explored data parallelism in lexing. We designed the first work-efficient parallel algorithm for finite state machines (FSMs), improving on the work-inefficient algorithm of Hillis and Steele [5] with novel algorithm-level speculation. The basic idea is to partition the input string among n processors. The problem is from which FSM state should a processor start scanning its string segment. Our empirical observation was that, in lexing, the automaton arrives to a stable state after a small number of characters, so it is often sufficient to prepend to each string segment a small suffix of its left neighbor [1]. Once the document has been so partitioned, we obtain n independent lexing tasks which can be vectorized with the help of a gather operation to allow simultaneous reads from a lookup table. On the CELL processor, our algorithm scales perfectly at least to six cores. To parallelize parsing [14], we have observed that old simple algorithms are a more promising starting point because, unlike the LR parser family, they have not been optimized with sequential execution in mind.

6 Parallel Page Layout

A page is laid out in two steps. Together, these steps account for half of the execution time in IE8 [13]. In the first step, the browser associates DOM nodes with CSS style rules. A rule might state that if a paragraph is labeled *important* and is nested in a box labeled *second-column*, then the paragraph should use a red font. Abstractly, every rule is predicated with a regular expression; these expressions are matched against node names, which are paths from the node to the root. Often, there are thousands of nodes and thousands of rules. This step may take 100–200ms on a laptop for a large page, and a magnitude longer on a handheld.

Matching a node against a rule is independent from others such matches. Per-node task partitioning thus exposes 1000-way parallelism. However, this algorithm is not work-efficient because tasks redo common work. We have explored caching algorithms and other sequential optimizations, achieving a 6-fold speedup vs. Firefox 3 on the Slashdot home page, and then gaining, from locality-aware task parallelism, another 5-fold speedup on 8 cores (with perfect scaling up to three cores). We are now exploring SIMD algorithms to simultaneously match a node against multiple rules or vice versa.

The second step lays out the page elements according to the styling rules. Like $\TeX$, CSS uses *flow* layout. Flow layout rules are inductive in that an element's position is computed after the preceding elements have been laid out. Layout engines thus perform (mostly) an in-order walk of the DOM. To parallelize this sequential traversal, we formalized a kernel of CSS as an attribute grammar and reformulated the layout into five tree traversal passes, each of which permits parallel handling of its heavy leaf tasks (which may involve font computations). Some node styles (*e.g., floats*) do not permit breaking sequential dependencies. For these nodes, we utilize speculative parallelization. We evaluated this algorithm with a model that ascribes uniform computation times to document nodes for every phase; task parallelism (with work stealing) accelerated the baseline algorithm 6-fold on an 8-core machine.

7 Parallel Scripting

Here we discuss the rationale for our scripting language. We start with concurrency issues in JavaScript programming and the parallelization that we expect to be beneficial. We then outline our actor language and discuss it briefly in the context of an application.

Concurrency JavaScript offers a simple concurrency model: there are neither locks nor threads, and event handlers (callbacks) are executed atomically. If the

DOM is changed by a callback, the page layout is recomputed before the execution handles the next event. This atomicity restriction is insufficient for preventing concurrency bugs. We have identified two sources of concurrency in browser programs—animations and interactions with web servers. Let us illustrate how this concurrency may lead to bugs in the context of GUI animations. Consider a mouse click that makes a window disappears with a shrinking animation effect. While the window is being animated, the user may keep interacting with it, for example by hovering over it, causing another animation effect on the window. The two animations may conflict, for example by simultaneously modifying the dimensions of the window. Ideally, the language should allow avoidance or at least the detection of such "animation race" bugs.

Parallelism and shared state To improve responsiveness of the browser, we need to execute atomic callbacks in parallel. To motivate such parallelization, consider programmatic animation of hundreds of elements. Simultaneous animation of many graphical elements is common in interactive data visualization; serializing these animations would reduce the frame rate of the animation.

The browser is allowed to execute callbacks in parallel as long as the observed execution appears to handle the events atomically. To define the commutativity of callbacks, we need to define shared state. Our preliminary design aggressively eliminates most shared state by exploiting properties of the browser domain. Actors can communicate only through message passing with copy semantics (*i.e.*, pointers and closures cannot be sent).

The shared state that may serialize callbacks comes in three forms. First, dependences among callbacks are induced by the DOM. A callback might resize a page component a, which may in turn lead the layout engine to resize a component b. A callback listening on the changes to the size of b is executed in response. We discuss below how we plan to detect independence of callbacks.

The second shared component is a local database with a relational interface. We desire a naming scheme for relations that allows inexpensive detection of script dependence, but have not yet decided on one; a static hierarchical name space may be sufficient. Efficiently implementing general data structures on top of a relational interface appears as hard as optimizing SETL programs, but we hope that the web scripting domain comes with a few idioms for which we can specialize.

The third component is the server. If a server is allowed to reorder responses, as is the case with web servers today, it appears that it cannot be used to synchronize scripts, but concluding this definitely for our programming model requires more work.

The Scripting Language Both parallelization and concurrency bug detection require analysis of dependences among callbacks. Analyzing JavaScript is however complicated by its programming model, which contains several "goto equivalents". First, data dependences among scripts in a web page are unclear because scripts can communicate indirectly through DOM nodes. The nodes are named with strings values that may be computed at run time; the names convey no structure and need not even be unique, complicating dependence analysis. Second, the flow of control is similarly obscured: reactive programming is implemented with callback handlers; while it may be possible to analyze callback control flow with flow analysis, the short-running scripts may prevent the more expensive analyses common in Java VMs.

We illustrate JavaScript programming with an example. The following program draws a box that displays the current time and follows the mouse trajectory, delayed by 500ms. The script references DOM elements with the string name `"box"`; the challenge for parallelization is to prove that the scripts modify the tree only locally. The example also shows the callbacks for the mouse event and for the timer event that draws the delayed box. These callbacks bind, rather obscurely, the mouse and the moving box; the challenge for the compiler is to determine which parts of the DOM tree are unaffected by the mouse so that they can be handled in parallel with the moving box.

```
<div id="box" style="position:absolute;">
      Time: <span id="time"> [[code not
          shown]] </span>
</div>
<script>
document.addEventListener (
   'mousemove',
   function (e) {  // called whenever the
        mouse moves
     var left = e.pageX;
     var top  = e.pageY;
     setTimeout(function() { // called 500
         ms later
       document.getElementById("box").
           style.top = top;
       document.getElementById("box").
           style.left = left;
     }, 500);
   }, false);
</script>
```

To simplify analysis of callback dependences, we need to make the following more explicit: (1) DOM node naming, so that we can what do we write in the DOM (1) dependences via layout;

To identify DOM names at JIT compile time, we propose static, hierarchical naming of DOM elements. We hope that the names can be obtained syntactically, rather than through type inference. This would allow the JIT compiler to discover during parsing whether two scripts operate on independent subtrees of the document; if so, the subtrees and their scripts can be placed on separate cores.

To prove that scripts commute, we will rely on the static structural naming of DOM elements, which allows us to detect if scripts reference a common subtree.

To account for dependences effected by the layout engine (see "Parallelism and shared state"), the HTML compiler will place a bound the changes in the DOM due to incremental relayout. This way, we will know that a change in a DOM leaf cannot propagate outside a particular DOM subtree, perhaps because the dimensions of that subtree is fixed by the web page designer. This will allow scripts to operate in parallel on multiple DOM subtrees.

To make the flow of control analyzable, we make the flow of data explicit, inspired by the FlapJax language [10]. The sources of dataflow are events such as the mouse and server responses, and the sinks are the DOM nodes. Below, we show the same program as an actor program. The program is now freed from the plumbing of low-level DOM manipulation and callback setup. Unlike the FlapJax language, we avoid mixing dataflow and imperative programming models at the same level; imperative programming will be encapsulated by actors that share no state (except for the DOM and a local database).

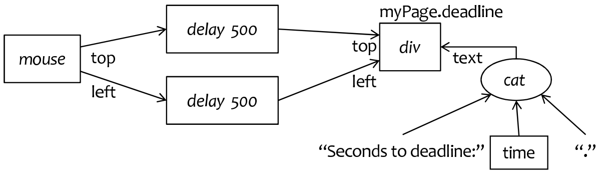

Figure 2: The same program as an actor program.

A Document as an Actor Network We described scripts as actors attached to the DOM: actors receive DOM events and, as output, create new ones, like setting DOM element attributes such as size and position (Figure 2). We now extend the use of actors in our semantic models: DOM elements are actors that form a DOM tree and collaborate on tasks like laying out the page. For example, an image nested in a paragraph sends to the paragraph its pixels so that the paragraph can prepare the laid-out paragraph pixel buffer. During layout computation, the image may influence the size of the paragraph or the image or vice versa, depending on styling rules. Browsers already treat frames from different URLs as actors; we just increase the granularity.

This extension makes the model/view partitioning more uniform. The *view* domain contains the DOM, layout, and animations rules and actors, and the *model* domain contains application logic actors. What are the benefits of this uniformity? First, the layout system will be able to reuse back-end infrastructure for parallel scripts and even allow user scripts. Second, once layout rules are programmable, it is easier to extend or modify the CSS layout model to fit an application's needs. For example, a math-mode layout system might be plugged in. Finally, since a node's display is constructed under programmer control, richer transforms will be possible, such as non-linear fish-eye zooms actings on true pixels. This task is hard in today's browsers where DOM nodes do not allow scripts to perform pixel-level introspection of display frames nor efficient manipulations of displays.

We discuss the culmination of our ideas in a hypothetical application, Mo'lympics, for viewing Olympic events, that we have been prototyping. An event consists of a video feed, basic information, and "rich commentary" made by others watching the same feed. Video segments can be recorded, annotated, and posted into the rich commentary. This rich commentary continuously updates as both professional analysts and home viewers post new information. A commentary might be another video, an embedded web page, or an interactive visualization. To view multiple events in Mo'lympics, events can be moved and zoomed in on.

Our preliminary design appears to be implementable with actors that have limited interaction through explicit, strongly typed message channels. The actors do not require shared state; surprisingly, message passing between actors is sufficient.

In the model domain, messages are user interface and network events, which are on the order of 1000 bytes. These can be transferred in around 10 clock cycles in today's STI Cell processor [11] and there are compiler optimizations to improve the locality of chatty actors. When larger messages must be communicated (such as video streams or source code), they should be handled by specialized browser services (video decoder and parser, respectively). It is the copying semantics of messages that allows this; such optimizations are well-known [4].

In the view domain, actors appear to transfer large buffers of rendered display elements through the view domain. However, this is a semantic notion; a view domain compiler could eliminate this overhead given knowledge of linear usage. This is another reason for separating the model and view domains: specialized compilers can make more intelligent decisions.

8 Related work

We draw linguistic inspiration from the Ptolemy project [2]; functional reactive programming, especially the FlapJax project [10]; and Max/MSP and LabVIEW.

We share systems design ideas with the Singularity OS project [7] and the Viewpoints Research Institute [8].

Parallel algorithms for parsing are numerous but mostly for natural-language parsing [14] or large data sets; we are not aware of efficient parallel parsers for computer languages nor parallel layout algorithms that fit the low-latency/small-task requirements of browsers.

Handhelds as thick clients is not new [12]; current application platforms pursue this idea (Android, iPhone, maemo). Others view handhelds as thin clients (DeepFish, SkyFire).

9 Summary

Web browsers could turn handheld computers into laptop replacements, but this vision poses new research challenges. We mentioned language and algorithmic design problems. There are additional challenges that we elided: scheduling independent tasks and providing quality of service guarantees are operating system problems; securing data and collaborating effectively are language, database, and operating system problems; working without network connectivity are also language, database, and operating system problems; and more. We have made promising steps towards solutions, but much work remains.

Acknowledgements 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 (also) 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.

References

[1] R. Bodik, C. G. Jones, R. Liu, L. Meyerovich, and K. Asanović. Browsing web 3.0 on 3.0 watts, 2008. Accessed 2008/10/19.

[2] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: a framework for simulating and prototyping heterogeneous systems. pages 527–543, 2002.

[3] J. Galenson, C. G. Jones, and J. Lo. Towards a Browser OS, December 2008. CS262a Course Project, UC Berkeley.

[4] A. Ghuloum. Ct: channelling nesl and sisal in c++. In *CUFP '07: Proceedings of the 4th ACM SIGPLAN workshop on Commercial users of functional programming*, pages 1–3, New York, NY, USA, 2007. ACM.

[5] W. D. Hillis and J. Guy L. Steele. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, 1986.

[6] M. Horowitz, E. Alon, D. Patil, S. Naffziger, R. Kumar, and K. Bernstein. Scaling, power, and the future of cmos. In *IEEE International Electron Devices Meeting*, Dec 2005.

[7] G. C. Hunt and J. R. Larus. Singularity: rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, 2007.

[8] A. Kay, D. Ingalls, Y. Ohshima, I. Piumarta, and A. Raab. Steps toward the reinvention of programming. Technical report, National Science Foundation, 2006.

[9] C. Lemuet, J. Sampson, J. Francois, and N. Jouppi. The potential energy efficiency of vector acceleration. *SC Conference*, 0:1, 2006.

[10] L. Meyerovich, M. Greenberg, G. Cooper, A. Bromfield, and S. Krishnamurthi. Flapjax, 2007. Accessed 2008/10/19.

[11] D. Pham, T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P. Harvey, H. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D. Stasiak, M. Suzuoki, O. Takahashi, J. Warnock, S. Weitzel, D. Wendel, and K. Yazawa. Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor. *Solid-State Circuits, IEEE Journal of*, 41(1):179–196, Jan. 2006.

[12] T. Starner. Thick clients for personal wireless devices. *Computer*, 35(1):133–135, 2002.

[13] C. Stockwell. What's coming in ie8, 2008. Accessed 2008/10/19.

[14] M. P. van Lohuizen. Survey of parallel context-free parsing techniques. Parallel and Distributed Systems Reports Series PDS-1997-003, Delft University of Technology, 1997.

[15] B. Zhai, R. G. Dreslinski, D. Blaauw, T. Mudge, and D. Sylvester. Energy efficient near-threshold chip multi-processing. In *ISLPED '07: Proceedings of the 2007 international symposium on Low power electronics and design*, pages 32–37, New York, NY, USA, 2007. ACM.