Parallelizing the Web Browser

Christopher Grant Jones, Rose Liu, Leo Meyerovich, Krste Asanović and Rastislav Bodík
Department of Computer Science, University of California, Berkeley

Abstract

We argue that the transition to handheld computers, predicted by Bell's Law of computer classes, will happen only if we invent new algorithms and change how we write software. While the previous computer classes could reuse the software of their ancestors, the energy efficiency for handheld operation may come from data parallelism in tasks that are currently sequential.

A web browser could become the dominant framework for deploying handheld applications, but it's hard to make the browser energy efficient. We show that the browser is CPU-bound and argue that it needs to be significantly parallelized. We examine parallelism in the browser as a mechanism for improving (1) energy efficiency and (2) user responsiveness. The former aspect calls for vectorizable parallel algorithms for parsing and page layout. The latter motivates a new web scripting programming model with linguistic support for the interaction between scripting and layout.

1 Browsers on Handheld Computers

Bell's Law of computer classes predicts that handheld computers will replace laptops. Internet-enabled phones have already eclipsed laptops in number, and may do so soon in functionality: The first phone with a pico-projector (Epoq) launched in summer 2008 and foldable displays are in the prototyping phase. Sonic pens (Mimiopad) can turn the phone into a tablet computer and statistical text entry (Dasher) promises to exploit the pen as a keyboard replacement.

These devices may give us a sufficient human interface, so why are iPhone and Android not vying to replace laptops? A key obstacle is the sluggishness of the mobile web browser. Perhaps surprisingly, browsers are CPU-bound: on the same network, the iPhone browser is 5 to 10-times slower than Firefox on a fast laptop, which makes iPhone unsuitable for anything but emergency browsing (for example, Slashdot takes 30 seconds to load). For Firefox, halving CPU frequency doubles the load times for cached pages; on the cold cache, the browser is still slowed down, by entire 50%.

The browser is sensitive to processor performance because it is simultaneously a compiler (for HTML), a page layout engine (for CSS), and an interpreter (for JavaScript). These are short-running tasks, but they are on the critical path of the user experience.

The performance of the mobile browser impacts software development. On the laptop, the browser has grown into a favorite application platform. Not so on the handheld: iPhone applications are native and web sites cripple their pages to match the mobile browser's computational capabilities. The trends do not bode well for the inferior mobile platform: new mobile processors will improve performance but browsers will host increasingly richer internet applications, such as gmail.

In summary, for many future applications on the laptop, the browser may be the platform of choice. We discuss in this paper how parallelism can make this possible on the handheld, too.

2 What Kind of Parallelism?

The performance of the browser is ultimately limited by energy constraints, which dictate how we ought to parallelize the browser. Ideally, we want to minimizes energy consumption while being sufficiently responsive. This motivates these questions:

- *Amount of parallelism:* Is it better to break down the browser into 10 or 1000 parallel computations?
- *Type of parallelism:* Should we exploit task parallelism, data parallelism, or both?
- *Algorithms:* What parallel algorithms are most energy efficient?

The energy equation for CMOS transistors tells us that maximum efficiency comes from many simple cores, operating at a slower frequency and lower supply voltage. In the 65nm Intel Tera-scale processor, reducing the frequency 4-times (and correspondingly reducing supply voltage) reduces the energy per operation 4-times; for the ARM9, reducing supply voltage to near-threshold levels and running on 16 cores, rather on one, reduces energy consumption by 80% [15].

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 [8]. We show that at least a part of the browser can be implemented in data parallel fashion.

Parallelism is thus an energy reduction technique. But how much parallelism do we need to uncover in the browser? For lack of precise analytical models, let us assume that a sustained 20-way browser parallelization

yields a sufficiently efficient execution. An interesting question now is how much of the browser can remain sequential to sustain the 20-way parallelism. Let us assume that the processor has 100 thin cores. Amdahl's Law tells us that only 4% of the browser execution can remain sequential, with the rest running at 100-way parallelism. Obtaining this level of parallelism is challenging because today's browser is optimized with techniques for single-threaded execution.

Another concern is cache-coherent shared memory. While it is not know how much one can save by replacing cache coherence with dedicated message passing support, a lower bound is 10–25% [9] but a factor of 2 to 3 seems plausible.

Parallel algorithms applied for energy efficiency must be close to work efficient—i.e., perform no more total work than a sequential algorithm—or else parallelization is counterproductive. The same argument applies for optimistic parallelization. Work efficiency is a demanding requirement since for some "inherently sequential" problems, like finite-state machines, only work-inefficient algorithms are known [4].

In summary, we want to decompose the browser into at least 100-way parallelism, in a combination of task and data parallelism with a preference for latter. Ideally, we will communicate with message passing and develop work-efficient algorithms.

3 The Browser Anatomy

Originally, web browsers were designed to render hypelinked documents. Soon, Java applets added interactivity. Later, JavaScript programs were allowed to modify the document itself, enabling animated menus and graphs. Today, scripts implement AJAX-style rich Internet applications (RIAs) that rival desktop programs.

The typical browser architecture 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). Content inlined by URLs from within the DOM is fetched. When all content necessary to display the page is present, and styling rules have been applied to the DOM, the page is laid out and drawn to the screen. (Good browsers actually lay out and draw a partially downloaded page.) After the initial page load, scripts respond to events generated by user input and server messages, and update the DOM accordingly. This, in turn, may cause the page's layout to change, and parts of the page to be rendered again.

4 Parallelizing the Browser

We consider the frontend of the browser separately from the scripting and the layout because the two require different parallelization.

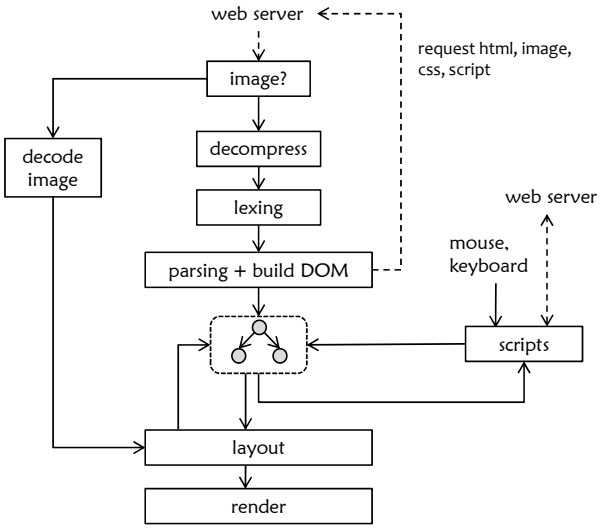

Figure 1: The architecture of today's browser.

4.1 Frontend

The browser frontend compiles (1) a document markup into its object model; (2) scripts into a more efficient executable format; and (3) style sheets into rules to be applied to the document's model. Internet Explorer 8 spends about 3-10% of its time in parsing-type tasks [13]. Our measurements show that Firefox spends up to 40in parsing. Both results surpass the amount of computation that we can execute sequentially. Task-level parallelization can be achieved by parsing downloaded files in parallel; unfortunately, a page usually has a small number of large files. We can produce more parallelism by pipelining the frontend; some problems remain, as JavaScript definition prevents the separation of lexing and parsing.

To uncover more parallelism, we have explored data parallelism in the lexical analysis, with the goal of parallelizing the processing of a single file. Thanks to a novel algorithmic speculation, we designed the first work-efficient parallel algorithm for FSMs, improving on the work-inefficient algorithm of Hillis and Steele [4]. The basic idea is to first partition the input string among n processors. The problem is from which FSM state should a processor start scanning the string segment? Our observation was that, at least in lexing, the automaton gets to a stable state after a small number of characters, so it is sufficient to prepend to each string segment a small part of its left neighbor [1]. Once the document has been so partitioned, we obtain n independent lexing tasks which get be solved in a vector fashion with a gather operation. In parallel parsing [14], we have observed that old simple algorithms are more promising because, like say LALR, they have not been optimized with sequential tricks.

4.2 Scripting and Page Layout

Rather than parallelizing JavaScript programs, we argue for an actor language with implicit parallelism. We believe that adding minimal shared state will suffice for the web domain and create a language accessible to web programmers. To support parallel execution, we compile away the slow scripting-layout interface by adding linguistic support for layout constraints.

Web Scripting Parallelizing JavaScript is challenging because its programming model is full of "goto equivalents." Data dependences between scripts in a web page are unclear because scripts can communicate through DOM nodes, which are named with strings that may be computed at run time; the names convey no structure and don't even need to be unique. The flow of control is similarly obscured: reactive programming is implemented by callbacks, akin to interrupt handlers. One could perhaps understand control flow with flow analysis, but the short-running scripts may prevent analyses common in JVMs.

JavaScript programming is illustrated in this example, which creates a box that displays the current time and follows the mouse trajectory, delayed by 500ms delay. Note how the script references DOM elements with the name (`"box"`); the two nested functions are callbacks for the mouse event and for the timer event. To keep things simple, we are forced to omit the code that displays the current time. The program obscures its functionality: how would you parallelize the animation of multiple boxes on the screen?

```
<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 500ms later
      document.getElementById("box").style.top = top;
      document.getElementById("box").style.left = left;
    }, 500);
  }, false);
</script>
```

We propose to support parallel script execution with a hierarchical naming scheme enforceable at JIT compile time. We hope that the names can be obtained syntactically, not through complex type inference. The compiler could thus discover during parsing whether two scripts operate on independnet subtrees of the document. To make the flow of control analyzable, we make the flow of data explicit. The sources of dataflow are events such as the mouse and server responses, and the sinks are the DOM nodes. We are inspired by the FlapJax language [10], but we avoid mixing programmign models; imperative programming will be all encapsulated in the actors who will share no state (we will say more on shared state shortly). We show below the same program as an actor program. Freed from the plumbing of low-level DOM manipulation and callback setup, the program is so compact that we can express completely how the time is computed.

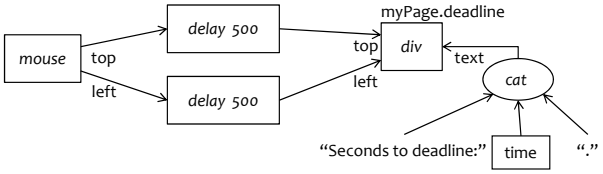

Figure 2: The same program as an actor program.

The Scripting-Layout Interface Page layout, often the browser bottleneck, poses three problems. First, specifications for real-world layout (CSS, TEX) are sequential: they prescribe, inductively, how to add a page element to preceding elements, already positioned on the page. We know of no constraint-based specifications and also no parallel layout algorithms. (See our separate submission on understanding the parallelism in CSS and on parallel algorithms for efficient layout computation.)

In this submission, we focus on the other two problems: the interpretive overhead and efficient uncovering of the parallelism. The interpretive overhead arises because the DOM is reinterpreted by the layout engine each time a script manipulates it. We will compile away the DOM: the scripts will feed values directly into the expression evaluated by the parallel solver. To uncover the parallelism during HTML compilation, we are formulating the layout problem as a hierarchical constraint system. The compiler will produce both the hierarchy of constraints, indicating independent layout subproblems, and also a constraint evaluation schedule.

The following web page fragment illustrates how our language relies on lexical scoping to produce constraint layout hierarchies. The fragment shows a box with vertically stacked input field and a horizonal box. The latter box displays the server's response to a user query.

```
VBOX {
  INPUT myInput { // declarative layout constraints
    // 'this' refers to 'myInput'
    width == (this.text.length+1)*1em
    border == this.isFocused ? min(4px, .2em) : 0
  }
  HBOX { // this script pipes together three actors:
    // read the query, format it and query the server
    myInput | \x.'http://google?word='+x | jsonRequest
  }
}
```

This fragment is compiled into constraints over attributes of page elements.

```
myInput.width = (myInput.text.length+1)*1em
```

In addition to explicit constraints, there are implicit constraints induced by the global web page styling rules:

```
vbox1.width = max(myInput.width+myInput.border,hbox1.
     width+hbox1.width)
```

Lexical scoping ensures that an element's rules refer only to parent's, children's and sibling's attributes, creating idnependennt subtrees. The compiler will also discover that widths and heights can be solved independently.

The Concurrency Model and Parallel Execution Web pages are not written by experts; threads and locks are out of the question. In our concurrency model, scripts listen for and respond to a sequence of events (such as user input, server responses, or changes to the DOM made by other scripts, the layout engine, or the frontend during page load). When an event arrives, a script is invoked and executed to termination; if the DOM is changed, its layout is recomputed, *e.g.* by updating it with new positions, and then rendered to the screen. The execution then handles the next event.

The browser is allowed to execute scripts in parallel as long as the observed execution appears to obey the above serial semantics. (We are similarly interested in executing two layout tasks simultaneously.) To define the commutativity of scripts, we need to define shared state. Our preliminary design aggressively eliminates most shared state, to explore the requirements of this domain and the abilities of linguistic and compiler support. The actors can communicate only through message passing with copy semantics (*i.e.*, pointers cannot be sent). The shared state has three components. First, dependences among scripts are induced by the DOM. For example a script may resize a page component a, which may in turn lead the layout engine to resize a component b. A script may be listening on the changes to the size of b so that it can accordingly adapt b's UI design or video playback rate. To prove that scripts commute, we rely on the static structural naming of DOM elements, which allows us to detect if scripts reference a common subtree. We also need to account for the indirect effects of a script, caused by the subsequent layout. For that, the HTML compiler will bound the changes in the DOM due to incremental relayout.

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. Instead, a static name space hierarchy similar to that of the DOM 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

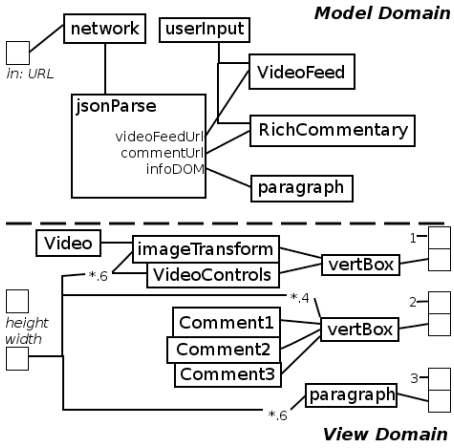

Figure 3: Actor decomposition of `OlympicEvent`.

scripting domain comes with a few idioms for which we can specialize.

Third, scripts may share a web server. Two scripts commute if they do not rely on our serial semantics to order requests to the server. A satisfactory solution may be to reorder requests originating from parallel scripts into the serial script execution order. If the network protocol is unordered, *e.g.* UDP, we can dispense with this check.

Finally, let us discuss parallel layout. Since the result of the page layout is rendered to the screen, serializing layout executions may significantly serialize scripts executions. Therefore, it makes sense to relax the semantics on what behavior is considered (literally) observable by the user on the screen. In particular, if two parts of the page are independent in the sense that their changes do not resize or otherwise reformat each other, their respective layouts can commute. This allows parallel layout and rendering of page fragments.

4.3 Realistic Example: Future Application

In this section, we show the culmination of our ideas in a real application: Mo'lympics, a future application for viewing Olympic events. We discuss the `OlympicEvent` actor show in Figure 3; please see [6] for a full description of Mo'lympics. `OlympicEvent` shows the video feed of a particular Olympic event, displays basic information about the event, and renders "rich commentary" made by others watching the same feed. Segments of video 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. The content of the commentary may be another playing video; a full, embedded web page; or an interactive visualization. We want to view multiple events in Mo'lympics, so we wish to be able to zoom in to and move around multiple `OlympicEvents`.

We hypothesize that `OlympicEvent` can be can be implemented by actors that have limited interaction through explicit, strongly typed message channels. We also think that actors should be divided into two domains: the *model* domain, containing the actors' logical hierarchy and application logic, and the *view* domain, containing its display hierarchy and layout and animation logic. We think that message passing is sufficient to capture most interaction between actors, so that shared memory will mostly be unnecessary. The actors in `OlympicEvent` do not require shared state; message passing along with scratchpad memory for each actor's local state is sufficient. These messages are mainly user interface and network events, which should be of a size on the order of 1000 bytes; these can be transferred in on the order of 10 clock cycles in today's STI Cell processor [11]. Compiler optimizations to improve the locality of chatty actors are also possible. When larger messages appear to be transferred (video streams in `VideoFeed` and the source code of pages in `RichCommentary`), in reality they would be handled by core browser services (video decoder and parser, respectively) and the apparent overhead could be compiled away. It is the copying semantics of messages that allows this; these optimizations are well known [3]. We will store nontrivial shared state in a local, concurrent database, exposed through an actor. One can imagine Mo'lympics using this to store storing user preferences or favorite videos. 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, perhaps by storing the rendered elements in a video buffer and manipulating them through graphics code, again possible because of copying semantics. This is another reason for separating the model and view domains: specialized compilers in each can make more intelligent decisions.

5 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 [5] and the Viewpoints Research Institute [7].

Parallel algorithms for parsing are numerous but mostly for natural-language parsing [14]; we not aware of work on efficient parallel parsing of computer languages and parallel layout algorithms.

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).

6 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 operating system and database problems; and more. We have made steps towards solutions, but much work remains.

Acknowledgements Research supported by Microsoft and Intel funding (Award # 20080469).

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] 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.

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

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

[6] C. G. Jones, L. Meyerovich, and R. Bodík. Berkeley parallel browser project blog, 2008. Accessed 2008/10/19.

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

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

[9] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis. Comparing memory systems for chip multiprocessors. *SIGARCH Comput. Archit. News*, 35(2):358–368, 2007.

[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 multiprocessing. In *ISLPED '07: Proceedings of the 2007 international symposium on Low power electronics and design*, pages 32–37, New York, NY, USA, 2007. ACM.