LEO MEYEROVICH

rethinking browser performance

Leo Meyerovich is a graduate student at the University of California, Berkeley, researching how better to write and run Web applications by exploiting programming languages. He created the Flapjax language for AJAX programming and helped start the Margrave project for understanding and verifying security policies. He is currently building a parallel Web browser.

lmeyerov@eecs.berkeley.edu

THE BROWSER WARS ARE BACK, AND performance may determine the winner this time around. Client-side performance is important enough to dictate the engineering of popular Web sites such as Facebook and Google, so the Web community is facing a crisis in balancing a high-level and accessible ecosystem on one side and squeezing out better performance on the other. Our research group has been reexamining core assumptions in how we architect browsers. The design space is surprisingly wide open, ranging from proxies in the cloud to multicore computers in our pockets.

Browser performance needs to be rethought. As part of the Berkeley Parallelism Lab, we are designing a new browser to achieve desirable Web application performance without sacrificing developer productivity. Our interest is forward-looking, including both future application domains, such as location-based services, and future hardware, such as multicore phones. One key axis of performance we have been investigating is how to incorporate parallelism from the bottom up [1]. We're not just settling on parallel algorithms: whether it's a sequential optimization or a new language, the community needs solutions.

Browser developers are facing a fascinating design space in trying to get the most out of our available cycles. As consumers of browsers, we benefit directly from their efforts, and as developers, we might find lessons for our own programs. In the following sections, after giving an idea about the role of performance in Web sites and current bottlenecks, we examine three fundamental axes for optimizing browsers. Understanding this design space motivates our own research in parallelizing browsers from the bottom up.

Why Performance?

Empirically, Web application speed impacts the bottom line for software developers. Near its release, tracking of Google Maps showed that site speedups were correlated to usage spikes. Similarly, faster return times of search queries increased usage. Photo-sharing sites have witnessed similar performance-driven phenomena, even if a typical user would not attribute their usage to it. Google's conclusion was that performance is important enough that they now factor page load time into their AdWords ranking algorithm! When the user experience directly affects sales, lowering performance barriers matters.

Should browser developers focus on performance? First, as seen with Google's actions, performance has a huge impact on user experience. Currently, developers are choosing between productivity and performance. Second, and more compelling, we have hit a wall with mobile devices. To be precise, we hit the power wall. Moore's Law holds, so transistors are still shrinking, but we cannot just keep clocking them up nor use them for further sequential hardware optimizations: because of energy (battery life) and power (heat) constraints, hardware architects switched focus to simpler but parallel architectures. For example, while a site such as Slashdot loads in three seconds on a laptop, it takes 17 seconds on an iPhone using the same wireless network. We expect performance to be the main decider in the handheld market. Investing in browser performance targets a common productivity drain and exposes emerging computing classes to more developers.

Bottlenecks

We first dispel the myth that the browser is network-bound. In a test of loading the top 25 popular US Web sites on various browsers, the IE8 team found the average total load time is 3.5–4 seconds, with 850 milliseconds being spent using the CPU [2]. Network traffic can often be taken off the critical path by smarter ordering or more careful caching and prefetching, and advances such as content delivery networks and mobile broadband are decreasing the actual network time. Unfortunately, the 850 milliseconds of computation is harder to explain away. Once we bring handhelds back into the picture, a 5–15x CPU slowdown becomes conspicuous. Table 1 details total page load times of popular Web sites on a MacBook Pro and an iPhone, measured by hand with a stopwatch when the Safari loading indicator stops. Note that the two devices use the same wireless network and all the Web sites are popular enough to be professionally optimized. To avoid caching phenomena, there were only 1–2 trials per site.*

** Wireless cards on laptops are superior to those on iPhones. The bandwidth difference is not sufficient to explain the performance disparity. Biasing the comparison in the other direction, the iPhone does not support Flash-based content such as advertisements.*

	slashdot.org	google.com	yahoo.com	wikipedia.org	myspace.com
MacBook Pro	3s	1s	1s	1s	2s
iPhone	17s	5s	14s	8s	15s

TABLE 1: TOTAL PAGE LOADING TIME FOR OPTIMIZED WEB SITES ON A LAPTOP AND HANDHELD USING A COLD CACHE AND THE SAME WIRELESS NETWORK

Where is the CPU time being spent? Despite the recent emphasis on faster JavaScript runtimes, on average, popular sites only spend 3% of their CPU time inside the JavaScript runtimes [3]. A JavaScript-heavy Web site such as a Webmail client will bump up the usage percentage to 14%; most of that time involves laying out the Web page and painting it, and, to a lesser extent, in more typical compiler front-end tasks like lexing and parsing. By Amdahl's Law, from font handling to matching CSS selectors, a lot needs to be faster.

We must also target future workloads. We predict increased usage of clientside storage, requiring a renewal in interest in speeding up structured personal storage. Scripts are also playing an increasing role, both in the number of interactions with browser libraries and in those with standalone components. Finally, we note a push toward more graphics: as augmented reality applications mature, such as Google Maps, Virtual Earth, and Photosynth, we expect the demand to grow even further, especially in the handheld space. Graphic accelerators have large power, energy, and performance benefits over thicker multicore solutions, so we even expect to see them in mobile devices, partially addressing how we expect to see at least one of these domains solved.

The Three Axes of Performance

We do not expect one silver bullet for browser performance, but, by systematically breaking down how performance can be improved, we have a basis for comparison and can understand the feasible extent of different approaches. Browsers are CPU-bound, so we should reanalyze how our CPU cycles are being spent. This leads to three fundamental axes for optimization.

AXIS 1: USE FEWER CYCLES

Can we get a desired job done with fewer operations? We break down this axis into three fundamental techniques:

- **Reduce functionality.** Phones have traditionally been under-provisioned, which has led to standards like WAP for writing applications with fewer features. Mobile versions of Web sites use the same idea: to ease the CPU load, Web site developers will simply remove functionality such as rich UIs. While this is an effective path to performance for application developers, for browser developers, the popular acceptance of this solution is a symptom of a systemic problem.
- **Avoid the abstraction tax.** Our group made a simple experiment: what happens if we naively reimplement Google Maps in C and thereby avoid the browser stack? We witnessed performance improvements of almost two magnitudes! While rewriting various libraries within browsers, we saw a similar trend: by more directly implementing components, skipping various indirection and safety layers, we observed drastic improvements.

  The community has latched onto this idea, leading to platforms like the iPhone SDK, Android, and Native Client or APIs like the canvas tag where developers code very close to the hardware. This is concerning. We do not want to give up the software engineering benefits of abstracting away hardware details and introducing productivity-related constructs. Furthermore, we do not want to sacrifice the Web ecosystem: programs such as search engines and browser extensions are largely flourishing because of the accessible, high-level structure of Web sites.
- **Optimize languages and libraries.** Ideally, we can shift the optimization burden to compiler and library developers. Interest in optimizing JavaScript has drastically increased, and a side benefit of rewriting layout engines to be standards-compliant has been to make them faster. However, while our experiences suggest there is a lot of room for sequential optimizations, the feasibility of developers of a multimillion-line codebase implementing fragile optimizations such as L1 cache tuning is unclear.

  Proebsting's observation about the alternative, compiler writers automating such optimizations, is worth recalling: once a language is reasonably implemented, compiler optimizations might yield 4% improvements a year, while hardware has given, on average, 60% [4]. We should chase magnitudes of improvement.

Developers are taking the first approach of simplifying their pages, and while we've been finding 5–70x improvements with the other approaches, they come at too high a cost.

AXIS 2: USE PARALLEL CYCLES

Even if we have exhausted our budget of sequential operations per second, we can follow Proebsting's Law and look towards exploiting hardware. Increases in performance will be largely through increased parallel operations per second. Given CMOS energy efficiency improvements of 25% per year, we expect about an additional core per device every year over the next decade, with each core supporting multiple hardware contexts and wide SIMD instructions.

Hardware advances have allowed us to largely reuse existing languages and libraries. Unfortunately, sufficient automatic parallelization has remained tantalizingly distant, even for functional and dataflow languages. It is not obvious that we can even manually parallelize programs such as browsers.

We note some concerns when parallelizing software such as browsers and ways such concerns are being assuaged:

- **Can browser libraries exploit parallelism?** Our group is methodically examining bottlenecks in browsers and parallelizing them, with our first result being for the canonically sequential FSM-like task of lexing. More significantly for browsers, we were able to design an algorithm to perform basic layout processing—determining the sizes and positions of elements on a page—in parallel, and are currently implementing it and iterating on its design. We are not alone in exploring this space. For example, video is already parallelized and we are not alone in rethinking parsing.
- **Can we exploit parallelism through concurrent calls?** We do not just want to parallelize the handling of individual calls into libraries. For example, can two different scripts interact with the layout library at the same time? Part of our process of designing new parallel libraries is to look out for such opportunities and think about how to detect the guarantees the libraries need to exploit them. For example, visually independent components, such as within <iframe> elements, correspond to actors whose layout computations are not dependent upon sibling elements.
- **Will parallelization make browsers more brittle?** To increase the integrity of browser runtimes, developers concerned with security have partitioned core libraries like the layout engine into OS processes, benefiting from address-space separation and management of resources such as CPU time. Much of our focus has been on libraries, where we have been using task-parallel systems such as TBB and Cilk++. This forces clearer code structure and interfaces. As a comparison, our sequential optimizations, like L1 cache optimizations, currently make code more brittle and inaccessible.
- **Given energy concerns, parallelization should be work-efficient.** A common trick in parallelization is to locally duplicate computations in order to avoid communication and synchronization overhead. Such tricks are not work-efficient, potentially wasting power and energy. Work efficiency means that if we were to simulate a parallel algorithm on a sequential computer, it should take the same amount of time as the sequential one. A common theme in our algorithms is *speculative execution:* we guess an invariant, process in parallel based on it, and patch up our computation as needed. For example, our layout algorithm speculates that one paragraph will not flow into the other, so they can be processed independently. The speculation is generally correct; when it is not, only the second paragraph needs to be recomputed. By bounding the recomputation, whether by localizing it or reducing its frequency, we approximate work efficiency.

We found some large yet simple opportunities for parallelism, such as with our new lexing and CSS selector algorithms. However, many other computations span large amounts of code (e.g., layout), and there is also a standing challenge in enabling Web designers to productively write parallel scripts such as animations.

AXIS 3: COMPUTE ELSEWHERE

If we cannot effectively exploit the cycles available on a personal device, perhaps we can use some elsewhere. For example, as latency decreases and bandwidth increases, a model like cloud computing becomes appealing. Web application developers already do this, by running database queries on servers, for example, and only UI computations on clients. Recently, we have witnessed reincarnations of X for browsers, allowing a thin client to display the results of running a browser elsewhere, or even just proxying individual plugins like Flash. It is worth reexamining how much computation we can (and should) redistribute. By this we primarily mean partitioning computations across different devices. It is also possible to partition over time. For example, search engines might cache popular queries, thick clients might prefetch content, and slower compilers often create faster bytecodes. User experience requirements combined with hardware constraints provide hints at the limits of offloading computation.

For the user experience, perceived latency is crucial. For example, browsers are now optimized to begin to display parts of a Web page before all of it is available, despite the cost of inducing extra computation. Perceived latency requirements vary by the type of task. Film—continuous, non-interactive motion—is generally shot at 24 frames per second, allowing 42ms to compute and render an animation. Many closed-loop systems, in which a user gets feedback while interacting with a system, such as by watching a mouse cursor move on a screen, have an upper bound of 100ms before tasks like verbally communicating or moving an object significantly suffer. For lower bounds, while hand tracking allows delays of 50–60ms, other domains are less forgiving (e.g., head-mounted displays with such long delays cause nausea). Finally, we note that there is a difference between delay and sampling rate: gestural and aural interactions should have samples processed with intervals on the order of milliseconds (and without jitter). For something like a hand drum with different strokes, both requirements are in place.

Considering end-to-end system latency costs, even when limited to network hops, it becomes clear that some computations are best when left on client devices for the foreseeable future. Consider a wireless device acting as a thin client for a proxied browser living in the cloud. From the device to a tower might be 10ms, and, looking forward, another 10ms from the tower to a hub. Round trip, that's 40ms already. Going between two hubs, such as LA and Seattle, on Internet2 is 40ms roundtrip (or 14ms at the speed of light); a proxy will only meet interactivity needs if we assume collocations to avoid this cost. Assuming a nearby hub, there is only 20ms round-trip latency, for a total 60ms network latency for a proxy. After that, we must consider device latency. We can add a delay of 10ms from an LCD, and an input device like a mouse might poll somewhere around every 5–10ms, bringing us to 75ms *without having done anything*. Even without including application-specific costs such as compressing/decompressing data for transmission or computing something with it (e.g., the animation or audio being interacted with), the space of proxyable content is already limited. Streaming a movie might be fine (with respect to latency), assuming highly tuned software, but user experience in other domains will already be subpar irrespective of the software. Forget turning your phone into a sensitive instrument.

There are further hardware concerns. In many locations and contexts, assuming fast Internet access, or even any access at all, is not possible. Another interesting cost is bandwidth. Browser use, in certain age groups, rivals TV use: proxying rich experiences has an associated bandwidth expense that must scale to support mainstream use. While TV streams might be shared between users, browsing sessions are more personal. Energy factors in again: proposals to increase bandwidth for devices, such as multiple antennas, are often still at the expense of battery life. Finally, we note that there are economic costs. Web server farms cache a lot of their computations, requiring little computation: as there will be less benefit from consolidating devices, much of the financial incentive of cloud computing disappears and a new pay model must close the gap. While we view latency as a dominating concern, energy, connectivity, cost, and bandwidth also have significant costs.

The situation is not entirely glum. Large computations should still be done on a server. Even interactive computations might be partitioned: for example, we experimented with a real-time mouse cursor, with positive results, but delayed scrollbar. Furthermore, breaking the browser experience out of the single device may still happen to enable new features, such as migrating a browsing session from a laptop to a phone when we leave the house or enabling remotely executing Flash scripts on today's slower handhelds. There is also the appeal of P2P systems, which may help boost bandwidth and lower latency, which we are beginning to study.

It seems that proxying solutions are best for larger or non-interactive experiences. It is not always clear when to make this distinction: for example, Gmail has shown that even though emails should be stored on the client, email search should be performed off-site. Finally, we note that computations that are too intensive for a client device will likely be performed in parallel, and, in a sense, they are probably even better suited for parallelization. Off-device computation will happen, but with many caveats. Understanding the standalone and integrated case is attractive, although we argue that the on-device case is emerging and should be exploited.

A Case Study: CSS Selectors

We recently examined CSS selectors, a pattern language for associating style rules with elements of a page. Developers use selectors to specify rules like "p.content a {font-style: italic}", meaning that links within content paragraphs should be italicized, where "p.content a" is the selector. When loading a large Web page with many style rules, such as Slashdot in Firefox, determining style constraints takes 100ms, with most of this spent in matching selectors. Selector speed has prompted David Hyatt, who worked on the CSS engines for Firefox and Safari as well as the overall language specification, to declare that the new "CSS3 selectors . . . really shouldn't be used at all if you care about page performance" [5]; indeed, tuned Web sites, like many by Google, do not use *any* selectors.

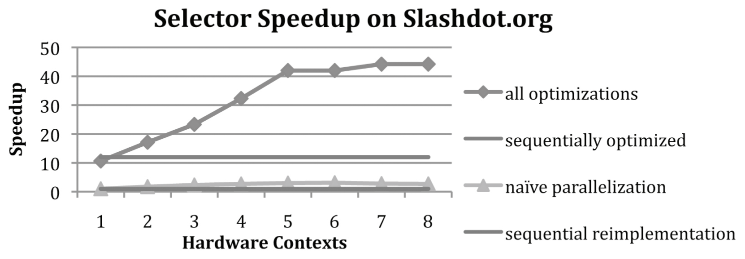

CHART 1: SPEEDUP OF CSS SELECTORS WHEN LOADING SLASHDOT.ORG

Over several months of on-and-off development, we implemented a new CSS selector engine from scratch. We started with the optimizations described in existing browsers and then advanced to our own sequential and parallel ones, until we achieved a matching time of 2ms on Facebook and Slashdot with an unoptimized pre-processing step of 5ms. Chart 1 shows the ultimate speedups from parallelizing the existing sequential algorithm (4x) vs. focusing on further sequential optimizations (11x) and then parallelizing (41x). The tests are on a 2.66GHz Nehalem prototype (two hardware threads per core and four cores on a socket). Not visible in the graph is how long it took to attain these optimizations: parallelization was significantly easier and, unlike with sequential optimizations, such as for better cache use, successive optimizations were generally complementary. Parallelization was only easy to an extent; the effort to go from one to four cores was less than that of going from four to five. Finally, we note that given the small size of these computations, it is not clear how to offload them to another device.

Conclusion

We are facing an exciting time of architectural transition. Productivity concerns involving high-level languages, large libraries, and software as a service are emerging as important enough to displace traditional low-level approaches. However, we are finding the need for much better performance, especially in the emerging computing class of handhelds. There is a lot of room for sequential optimizations, but as the opportunity cost for them is high, we instead advocate focusing more on exploiting hardware-driven optimizations. This is taking place in the form of local, parallel computations and networked (and still parallel) computations. Overall, we found parallelizing on-device browser computations to be the most enticing direction for improving performance.

ACKNOWLEDGMENTS

Professor Ras Bodik has been heading our browser group, with Chris Jones (now at Mozilla) as another founding member. Professor Krste Asanović, Shaoib Kamil, Kaushik Datta, Rajesh Nishtala, and Christopher Batten of the Parallelism Lab and Robert O'Callahan and David Mandelin at Mozilla have provided valuable advice throughout our work.

REFERENCES

[1] Chris Jones, Rose Liu, Leo Meyerovich, Krste Asanović, and Rastislav Bodik, "Parallelizing the Web Browser," *Proceedings of the 1st USENIX Workshop on Hot Topics in Parallelism* (HotPar '09), March 2009.

[2] Shreesh Dubey, "AJAX Performance Measurement Methodology for Internet Explorer 8 Beta 2," *CoDe Magazine* 5(3), 2008: http://www.code-magazine.com/Article.aspx?quickid=0811102.

[3] Microsoft, "Measuring Browser Performance: Understanding Issues in Benchmarking and Performance Analysis," 2009: http://www.microsoft.com/downloads/details.aspx?displaylang=en&FamilyID=cd8932f3-b4be-4e0e-a73b-4a373d85146d.

[4] Todd Proebsting, "Proebsting's Law": http://research.microsoft.com/en-us/um/people/toddpro/papers/law.htm.

[5] Shaun Inman, "CSS Qualified Selectors": http://www.shauninman.com/archive/2008/05/05/css_qualified_selectors.