Experience Report: Developing the Servo Web Browser Engine using Rust

[DRAFT]

Brian Anderson Lars Bergstrom
David Herman Josh Matthews
Keegan McAllister Jack Moffitt
Simon Sapin
Mozilla Research
{banderson,larsberg,dherman,jdm,
kmcallister,jack,simonsapin}@mozilla.com

Manish Goregaokar
Indian Institute of Technology Bombay
manishg@iitb.ac.in

Abstract

All modern web browsers — Internet Explorer, Firefox, Chrome, Opera, and Safari — have a core rendering engine written in C++. This language choice was made because it affords the systems programmer complete control of the underlying hardware features and memory in use, and it provides a transparent compilation model.

Servo is a project started at Mozilla Research to build a new web browser engine that preserves the capabilities of these other browser engines but also both takes advantage of the recent trends in parallel hardware and is more memory-safe. We use a new language, Rust, that provides us a similar level of control of the underlying system to C++ but which builds on many concepts familiar to the functional programming community, forming a novelty — a useful, safe systems programming language.

In this paper, we show how a language with an affine type system, regions, and many syntactic features familiar to functional language programmers can be successfully used to build state-of-the-art systems software. We also outline several pitfalls encountered along the way and describe some potential areas for future research.

1. Introduction

The heart of a modern web browser is the browser engine, which is the code responsible for loading, processing, evaluating, and rendering web content. There are three major browser engine families:

1. Trident/Spartan, the engine in Internet Explorer [IE]
2. Webkit[WEB]/Blink, the engine in Safari [SAF], Chrome [CHR], and Opera [OPE]
3. Gecko, the engine in Firefox [FIR]

All of these engines have at their core many millions of lines of C++ code. While the use of C++ has enabled all of these browsers to achieve excellent sequential performance on a single web page, on mobile devices with lower processor speed but many more processors, these browsers do not provide the same level of interactivity that they do on desktop processors [MTK$^+$12, CFMO$^+$13]. Further, in an informal inspection of the critical security bugs in Gecko, we determined that roughly 50% of the bugs are use after free, out of range access, or related to integer overflow. The other 50% are split between errors in tracing values from the JavaScript heap in the C++ code and errors related to dynamically compiled code.

Servo [SER] is a new web browser engine designed to address the major environment and architectural changes over the last decade. The goal of the Servo project is to produce a browser that enables new applications to be authored against the web platform that run with more safety, better performance, and better power usage than in current browsers. To address memory-related safety issues, we are using a new systems programming language, Rust [RUS]. For parallelism and power, we scale across a wide variety of hardware by building either data- or task-parallelism, as appropriate, into each part of the web platform. Additionally, we are improving concurrency by reducing the simultaneous access to data structures and using a message-passing architecture between components such as the JavaScript engine and the rendering engine that paints graphics to the screen. Servo is currently over 400k lines of Rust code and implements enough of the web to render and process many pages, though it is still a far cry from the over 7 million lines of code in the Mozilla Firefox browser and its associated libraries. However, we believe that we have implemented enough of the web platform to provide an early report on the successes, failures, and open problems remaining in Servo, from the point of view of programming languages and runtime research. In this experience report, we discuss the design and architecture of a modern web browser engine, show how modern programming language techniques — many of which originated in the functional programming community — address these design constraints, and also touch on ongoing challenges and areas of research where we would welcome additional community input.

2. Browsers

Modern web browsers do not just load static pages, but can also handle pages that have similar complexity to native applications.

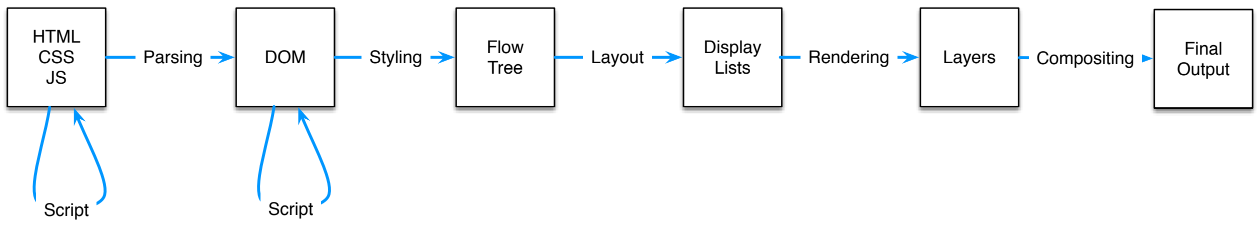

Figure 1. Processing stages and intermediate representations in a browser engine.

From application suites such as the Google Apps[1] to games based on the Unreal Engine[2], modern browsers have the ability to handle much more than simple static pages. Figure 1 shows the steps in processing a site. While the naming is specific to the Servo browser, similar steps are used in all modern browsers.[3]

2.1 Parsing HTML and CSS

A URL identifies a resource to load. This resource usually consists of HTML, which is then parsed and typically turned into a Document Object Model (DOM) tree. From a programming languages standpoint, there are several interesting aspects of the parser design for HTML. First, though the specification allows the browser to abort on a parse error[4], in practice browsers follow the recovery algorithms described in that specification precisely so that even ill-formed HTML will be handled in an interoperable way across all browsers. Second, due to the presence of the `<script>` tag, the token stream can be modified during operation. For example, the below example that injects an open tag for the header and comment blocks works in all modern browsers.

```
<html>
  <script>
  document.write("<h");
  </script>1>
  This is a h1 title

  <script>
  document.write("<!-");
  </script>-
  This is commented
  -->
</html>
```

This requires parsing to pause until JavaScript code has run to completion. But, since resource loading is such a large factor in the latency of loading many webpages (particularly on mobile), all modern parsers also perform speculative token stream scanning and prefetch of resources likely to be required [WLZC11].

2.2 Styling

After constructing the DOM, the browser uses the styling information in linked CSS files and in the HTML to compute a styled tree of flows and fragments, called the *flow tree* in Servo. This process can create many more flows than previously existed in the DOM — for example, when a list item is styled to have an associated counter glyph.

2.3 Layout

The flow tree is then processed to produce a set of *display list* items. These list items are the actual graphical elements, text runs, etc. in their final on-screen positions. The order in which these elements are displayed is well-defined by the standard[5].

2.4 Rendering

Once all of the elements to appear on screen have been computed, these elements are rendered, or painted, into memory buffers or directly to graphics surfaces.

2.5 Compositing

The set of memory buffers or graphical surfaces, called *layers*, are then transformed and composited together to form a final image for presentation. Layerization is used to optimize interactive transformations like scrolling and certain animations.

2.6 Scripting

Whether through timers, `<script>` blocks in the HTML, user interactions, or other event handlers, JavaScript code may execute at any point during parsing, layout, and painting or afterwards during display. These scripts can modify the DOM tree, which may require rerunning the layout and painting passes in order to update the output. Most modern browsers use some form of dirty bit marking to attempt to minimize the recalculations during this process.

3. Rust

Rust is a statically typed systems programming language most heavily inspired by the C and ML families of languages [RUS]. Like the C family of languages, it provides the developer fine control over memory layout and predictable performance. Unlike C programs, Rust programs are *memory safe* by default, only allowing unsafe operations in specially-delineated blocks.

Rust features an *ownership-based* type system inspired by the region systems work in the MLKit project [TB98] and especially as implemented in the Cyclone language [GMJ+02]. Unlike the related ownership system in Singularity OS [HLA+05], Rust allows programs to not only transfer ownership but also to temporarily *borrow* owned values, significantly reducing the number of region and ownership annotations required in large programs. The ownership model encourages immutability by default while allowing for controlled mutation of owned or uniquely-borrowed values.

Complete documentation and a language reference for Rust are available at: `http://doc.rust-lang.org/`.

[1] `https://apps.google.com`

[2] `https://www.unrealengine.com/`

[3] `http://www.html5rocks.com/en/tutorials/internals/howbrowserswork/`

[4] `https://html.spec.whatwg.org/multipage/#parsing`

[5] `http://www.w3.org/TR/CSS21/zindex.html#painting-order`

```
fn main() {
  // An owned pointer to a heap-allocated
  // integer
  let mut data = Box::new(0);

  Thread::spawn(move || {
    *data = *data + 1;
  });
  // error: accessing moved value
  print!("{}", data);
}
```

Figure 2. Code that will not compile because it attempts to access mutable state from two threads.

```
fn main() {
  // An immutable borrowed pointer to a
  // stack-allocated integer
  let data = &1;

  Thread::scoped(|| {
    println!("{}", data);
  });
  print!("{}", data);
}
```

Figure 3. Safely reading immutable state from two threads.

```
fn main() {
  // A heap allocated integer protected by an
  // atomically-reference-counted mutex
  let data = Arc::new(Mutex::new(0));
  let data2 = data.clone();

  Thread::scoped(move || {
    *data2.lock().unwrap() = 1;
  });

  print!("{}", *data.lock().unwrap());
}
```

Figure 4. Safely mutating state from two threads.

3.1 Ownership and concurrency

Because the Rust type system provides very strong guarantees about memory aliasing, Rust code is memory safe even in concurrent and multithreaded environments, but beyond that Rust also ensures *data-race freedom*.

In concurrent programs, the data operated on by distinct threads is also itself distinct: under Rust's ownership model, data cannot be owned by two threads at the same time. For example, the code in Figure 2 generates a static error from the compiler because after the first thread is spawned, the ownership of `data` has been transferred into the closure associated with that thread and is no longer available in the original thread.

On the other hand, the immutable value in Figure 3 can be borrowed and shared between multiple threads as long as those threads don't outlive the scope of the data, and even mutable values can be shared as long as they are owned by a type that preserves the invariant that mutable memory is unaliased, as with the mutex is Figure 4.

Site	Gecko	Servo 1 thread	Servo 4 threads
Reddit	250	100	55
CNN	105	50	35

Table 1. Performance of Servo against Mozilla's Gecko rendering engine on the layout portion of some common sites. Times are in milliseconds, where lower numbers are better.

With relatively few simple rules, ownership in Rust enables foolproof task parallism, but also data parallism, by partitioning vectors and lending mutable references into properly scoped threads. Rust's concurrency abstractions are entirely implemented in libraries, and though many advanced concurrent patterns such as work-stealing [ABP98] cannot be implemented in safe Rust, they can usually be encapsulated in a memory-safe interface.

4. Servo

A crucial test of Servo is performance — Servo must be at least as fast as other browsers at similar tasks to succeed, even if it provides additional memory safety. Table 1 shows a preliminary comparison of the performance of the layout stage (described in Section 2) of rendering several web sites in Mozilla Firefox's Gecko engine compared to Servo, taken on a modern MacBook Pro.

In the remainder of this section, we cover specific areas of Servo's design or implementation that make use of Rust and the impacts and limitations of these features.

4.1 Rust's syntax

Rust has struct and enum types (similar to Standard ML's record types and datatypes [MTHM97]) as well as pattern matching. These types and associated language features provide two large benefits to Servo over traditional browsers written in C++. First, creating new abstractions and intermediate representations is syntactically easy, so there is very little pressure to tack additional fields into classes simply to avoid creating a large number of new header and implementation files. More importantly, pattern matching with static dispatch is typically faster than a virtual function call on a class hierarchy. Virtual functions can both have an in-memory storage cost associated with the virtual fuction tables (sometimes many thousands of bytes[6]) but more importantly incur indirect function call costs. All C++ browser implementations transform performance-critical code to either use the `final` specifier wherever possible or specialize the code in some other way to avoid this cost.

Rust also attempted to stay close to familiar syntax, but did not require full fidelity or easy porting of programs from languages such as C++. This approach has worked well for Rust because it has prevented some of the complexity that arose in Cyclone [GMJ$^+$02] with their attempts to build a safe language that required minimal porting effort for even complicated C code.

4.2 Compilation strategy

Many statically typed implementations of polymorphic languages such as Standard ML of New Jersey [AM91] and OCaml [Ler00] have used a compilation strategy that optimizes representations of data types when polymorphic code is monomorphically used, but defaults to a less efficient style otherwise, in order to share code [Ler90]. This strategy reduces code size, but leads to unpredictable performance and code, as changes to a codebase that either add a new instantiation of a polymorphic function at a given type or,

[6] https://chromium.googlesource.com/chromium/blink/+/c048c5c7c2578274d82faf96e9ebda4c55e428da

in a modular compilation setting, that expose a polymorphic function externally, can change the performance of code that is not local to the change being made.

Monomorphization, as in MLton [CFJW], instead instantiates each polymorphic code block at each of the types it is applied against, providing predictable output code to developers at the cost of some code duplication. This strategy is used by virtually all C++ compilers to implement templates, so it is proven and well-known within systems programming. Rust also follows this approach, although it improves on the ergonomics of C++ templates by embedding serialized generic function ASTs within "compiled" binaries.

Rust also chooses a fairly large default compilation unit size. A Rust *crate* is subject to whole-program compilation [Wee06], and is optimized as a unit. A crate may comprise hundreds of modules, which provide namespacing and abstraction. Module dependencies within a crate are allowed to be cyclic.

The large compilation unit size slows down compilation and especially diminishes the ability to build code in parallel. However, it has enabled us to write Rust code that easily matches the sequential speed of its C++ analog, without requiring the Servo developers to become compiler experts. Servo contains about 600 modules within 20 crates.

4.3 Memory management

As described in Section 3, Rust has an affine type system that ensures every value is used at most once. One result of this fact is that in the more than two years since Servo has been under development, we have encountered zero use-after-free memory bugs in safe Rust code. Given that these bugs make up such a large portion of the security vulnerabilities in modern browsers, we believe that even the additional work required to get Rust code to pass the type checker initially is justified.

One area for future improvement is related to allocations that are not owned by Rust itself. Today, we simply wrap raw C pointers in **`unsafe`** blocks when we need to use a custom memory alloctor or interoperate with the SpiderMonkey JavaScript engine from Gecko. We have implemented wrapper types and compiler plugins that restrict incorrect uses of these foreign values, but they are still a source of bugs and one of our largest areas of unsafe code.

Additionally, Rust's ownership model assumes that there is a single owner for each piece of data. However, many data structures do not follow that model, in order to provide multiple traversal APIs without favoring the performance of one over the other. For example, a doubly-linked list contains a back pointer to the previous element to aid in traversals in the opposite direction. Many optimized hashtable implementations also have both hash-based access to items and a linked list of all of the keys or values. In Servo, we have had to use unsafe code to implement data structures with this form, though we are typically able to provide a safe interface to users.

4.4 Language interoperability

Rust has nearly complete interoperability with C code, both exposing code to and using code from C programs. This support has allowed us to smoothly integrate with many browser libraries, which has been critical for bootstrapping a browser without rewriting all of lower-level libraries immediately, such as graphics rendering code, the JavaScript engine, font selection code, etc. Table 2 shows the breakdown between current lines of Rust code (including generated code that handles interfacing with the JavaScript engine) and C code. This table also includes test code, though the majority of that code is in HTML and JavaScript.

There are two limitations in the language interoperability that pose challenges for Servo today. First, Rust cannot currently expose varargs-style functions to C code. Second, Rust cannot compile against C++ code. In both cases, Servo uses C wrapper code to call into the code that Rust cannot directly reach. While this approach is not a large problem for varargs-style functions, it defeats many of the places where the C++ code has been crafted to ensure the code is inlined into the caller, resulting in degraded performance. We intend to fix this through cross-language inlining, taking advantage of the fact that both `rustc` and `clang++` can produce output in the LLVM intermediate representation [LLV], which is subject to link-time optimization. We have demonstrated this capability at small scale, but have not yet deployed it within Servo.

Language	Lines of Code
C or C++	1,678,427
Rust	410,817
HTML or JavaScript	217,827

Table 2. Lines of code in Servo

4.5 Libraries and abstractions

Many high-level languages provide abstractions over I/O, threading, parallelism, and concurrency. Rust provides functionality that addresses each of these concerns, but they are designed as thin wrappers over the underlying services, in order to provide a predictable, fast implementation that works across all platforms. Therefore, much like other modern browsers, Servo contains many of its own specialized implementations of library functions that are tuned for the specific cases of web browsers. For example, we have special "small" vectors that allow instantiation with a default inline size, as there are use cases where we create many thousands of vectors, nearly none of which have more than 4 elements. In that case, removing the extra pointer indirection — particularly if the values are of less than pointer size — can be a significant space savings. We also have our own work-stealing library that has been tuned to work on the DOM and flow trees during the process of styling and layout, as described in Section 2. It is our hope that this code might be useful to other projects as well, though it is fairly browser-specific today.

Concurrency is available in Rust in the form of CML-style channels [Rep91], but with a separation between the reader and writer ends of the channel. This separation allows Rust to enforce a multiple-writer, single-reader constraint, both simplifying and improving the performance of the implementation over one that supports multiple readers. We have structured the entire Servo browser engine as a series of threads that communicate over channels, avoiding unsafe explicitly shared global memory for all but a single case (reading properties in the flow tree from script, an operation whose performance is crucially tested in many browser benchmarks). The major challenge we have encountered with this approach is the same one we have heard from other designers of large concurrent systems — reasoning about whether protocols make progress or threads eventually terminate is manual and quite challenging, particularly in the presence of arbitrary thread failures.

4.6 Macros

Rust provides a hygienic macro system. Macros are defined using a declarative, pattern-based syntax [KW87]. The macro system has proven invaluable; we have defined more than one hundred macros throughout Servo and its associated libraries.

For example, our HTML tokenizer rules, such as those shown in Figure 5, are written in a macro-based domain specific language that closely matches the format of the HTML specification.[7] An-

[7] `https://html.spec.whatwg.org/multipage/syntax.html#tokenization`

```
match self.state {
  states::Data => loop {
    match get_char!(self) {
      '&'  => go!(self: consume_char_ref),
      '<'  => go!(self: to TagOpen),
      '\0' => go!(self: error; emit '\0'),
      c    => go!(self: emit c),
```

Figure 5. Incremental HTML tokenizer rules, written in a succinct form using macros. Macro invocations are of the form `identifier!(...)`.

other macro handles incremental tokenization, so that the state machine can pause at any time to await further input. If no next character is available, the `get_char!` macro will cause an early return from the function that contains the macro invocation. This careful use of non-local control flow, together with the overall expression-oriented style, makes Servo's HTML tokenizer unusually succinct and comprehensible.

The Rust compiler can also load compiler plugins written in Rust. These can perform syntactic transformations beyond the capabilities of the hygienic pattern-based macros. Compiler plugins use unstable internal APIs, so the maintenance burden is high compared to pattern-based macros. Nevertheless, Servo uses procedural macros for a number of purposes, including building perfect hash maps at compile time,[8] interning string literals, and auto-generating GC trace hooks. Despite the exposure to internal compiler APIs, the deep integration with tooling makes procedural macros an attractive alternative to the traditional systems metaprogramming tools of preprocessors and code generators.

4.7 Project-specific static analysis

Compiler plugins can also provide "lint" checks[9] that use the same infrastructure as the compiler's built-in warnings. This allows project-specific safety or style checks to integrate deeply with the compiler. Lint plugins traverse a typechecked abstract syntax tree (AST), and they can be enabled/disabled within any lexical scope, the same way as built-in warnings.

Lint plugins provide some essential guarantees within Servo. Because our DOM objects are managed by the JavaScript garbage collector, we must add GC roots for any DOM object we wish to access from Rust code. Interaction with a third-party GC written in C++ is well outside the scope of Rust's built-in guarantees, so we bridge the gap with lint plugins. These capabilities enable a safer and more correct interface to the SpiderMonkey garbage collector. For example, we can enforce at compile time that, during the tracing phase of garbage collection, all Rust objects visible to the GC will report all contained pointers to other GC values, avoiding the threat of incorrectly collecting reachable values. Furthermore, we restrict the ways our wrappers around SpiderMonkey pointers can be manipulated, thus turning potential runtime memory leaks and ownership semantic API mismatches into static compiler errors instead.

As the "lint" / "warning" terminology suggests, these checks may not catch all possible mistakes. Ad-hoc extensions to a type system cannot easily guarantee soundness. Rather, lint plugins are a lightweight way to catch mistakes deemed particularly common or damaging in practice. As plugins are ordinary libraries, members of the Rust community can share lint checks that they have found useful.[10]

Future plans include refining our safety checks for garbage collected values, such as flagging invalid ownership transference, and introducing compile-time checks for constructs that are non-optimal in terms of performance or memory usage.

5. Open problems

While this work has discussed many challenges in browser design and our current progress, there are many other interesting open problems.

Just-in-time code. JavaScript engines dynamically produce native code that is intended to execute more efficiently than an interpreted strategy. Unfortunately, this area is a large source of security bugs. These bugs come from two sources. First, there are potential correctness issues. Many of these optimizations are only valid when certain conditions of the calling code and environment hold, and ensuring the specialized code is called only when those conditions hold is non-trivial. Second, dynamically producing and compiling native code and patching it into memory while respecting all of the invariants required by the JavaScript runtime (e.g., the garbage collector's read/write barriers or free vs. in-use registers) is also a challenge.

Integer overflow/underflow. It is still an open problem to provide optimized code that checks for overflow or underflow without incurring significant performance penalties. The current plan for Rust is to have debug-only checking of integer ranges and for Servo to run debug builds against a test suite, but that may miss scenarios that only occur in optimized builds or that are not represented by the test suite.

Unsafe code correctness. Today, when we write unsafe code in Rust there is limited validation of memory lifetimes or type safety within that code block. However, many of our uses of unsafe code are well-behaved translations of either pointer lifetimes or data representations that cannot be annotated or inferred in Rust. We are very interested in additional annotations that would help us prove basic properties about our unsafe code, even if these annotations require a theorem prover or ILP solver to check.

Incremental computation. As mentioned in Section 2, all modern browsers use some combination of dirty bit marking and incremental recomputation heuristics to avoid reprocessing the full page when a mutatation is performed. Unfortunately, these heuristics are not only frequently the source of performance differences between browsers, but they are also a source of correctness bugs. A library that provided a form of self adjusting computation suited to incremental recomputation of only the visible part of the page, perhaps based on the Adapton [HPHF14] approach, seems promising.

6. Related browser research

The ZOOMM browser was an effort at Qualcomm Research to build a parallel browser, also focused on multicore mobile devices [CFMO+13]. This browser includes many features that we have not yet implemented in Servo, particularly around resource fetching. They also wrote their own JavaScript engine, which we have not done. Servo and ZOOMM share an extremely concurrent architecture — both have script, layout, rendering, and the user interface operating concurrently from one another in order to maximize interactivity for the user. Parallel layout is one major area that was not investigated in the ZOOMM browser, but is a focus of Servo. The other major difference is that Servo is implemented in

[8] `https://github.com/sfackler/rust-phf`

[9] `http://doc.rust-lang.org/book/plugins.html#lint-plugins`

[10] `https://github.com/Manishearth/rust-clippy`

Rust, whereas ZOOMM is written in C++, similarly to most modern browser engines.

Ras Bodik's group at the University of California Berkeley worked on a parallel browsing project (funded in part by Mozilla Research) that focused on improving the parallelism of layout [MB10]. Instead of our approach to parallel layout, which focuses on multiple parallel tree traversals, they modeled a subset of CSS using attribute grammars. They showed significant speedups with their system over a reimplementation of Safari's algorithms, but we have not used this approach due to questions of whether it is possible to use attribute grammars to both accurately model the web as it is implemented today and to support new features as they are added. Servo uses a very similar CSS selector matching algorithm to theirs.

Acknowledgments

The Servo project was started at Mozilla Research, but has benefitted from contributions from Samsung, Igalia, and hundreds of volunteer community members. These contributions amount to more than half of the commits to the project, and we are grateful for both our partners' and volunteers' efforts to make this project a success.

References

[ABP98] Arora, N. S., R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors, 1998.

[AM91] Appel, A. W. and D. B. MacQueen. Standard ML of New Jersey. In *PLIP '91*, vol. 528 of *LNCS*. Springer-Verlag, New York, NY, August 1991, pp. 1–26.

[CFJW] Cejtin, H., M. Fluet, S. Jagannathan, and S. Weeks. The MLton Standard ML compiler. Available at `http://mlton.org`.

[CFMO+13] Cascaval, C., S. Fowler, P. Montesinos-Ortego, W. Piekarski, M. Reshadi, B. Robatmili, M. Weber, and V. Bhavsar. ZOOMM: A parallel web browser engine for multicore mobile devices. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, Shenzhen, China, 2013. ACM, pp. 271–280.

[CHR] The Google Chrome web browser. `http://www.google.com/chrome`.

[FIR] The Mozilla Firefox web browser. `http://www.firefox.com/`.

[GMJ+02] Grossman, D., G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, Berlin, Germany, 2002. ACM, pp. 282–293.

[HLA+05] Hunt, G., J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. D. Zill. An Overview of the Singularity Project. *Technical Report MSR-TR-2005-135*, Microsoft Research, October 2005.

[HPHF14] Hammer, M. A., K. Y. Phang, M. Hicks, and J. S. Foster. Adapton: Composable, demand-driven incremental computation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, Edinburgh, United Kingdom, 2014. ACM, pp. 156–166.

[IE] The Microsoft Internet Explorer web browser. `http://www.microsoft.com/ie`.

[KW87] Kohlbecker, E. E. and M. Wand. Macro-by-example: Deriving syntactic transformations from their specifications. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, Munich, West Germany, 1987. ACM, pp. 77–84.

[Ler90] Leroy, X. Efficient data representation in polymorphic languages. In P. Deransart and J. Małuszyński (eds.), *Programming Language Implementation and Logic Programming 90*, vol. 456 of *Lecture Notes in Computer Science*. Springer, 1990.

[Ler00] Leroy, X. *The Objective Caml System (release 3.00)*, April 2000. Available from `http://caml.inria.fr`.

[LLV] The LLVM compiler infrastructure. `http://llvm.org`.

[MB10] Meyerovich, L. A. and R. Bodik. Fast and Parallel Webpage Layout. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, Raleigh, North Carolina, USA, 2010. ACM, pp. 711–720.

[MTHM97] Milner, R., M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, MA, 1997.

[MTK+12] Mai, H., S. Tang, S. T. King, C. Cascaval, and P. Montesinos. A Case for Parallelizing Web Pages. In *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism*, HotPar'12, Berkeley, CA, 2012. USENIX Association.

[OPE] The Opera web browser. `http://www.opera.com/`.

[Rep91] Reppy, J. H. CML: A higher-order concurrent language. In *PLDI '91*. ACM, June 1991, pp. 293–305.

[RUS] The Rust language. `http://www.rust-lang.org/`.

[SAF] The Apple Safari web browser. `http://www.apple.com/safari`.

[SER] The Servo web browser engine. `https://github.com/servo/servo`.

[TB98] Tofte, M. and L. Birkedal. A region inference algorithm. *ACM Trans. Program. Lang. Syst.*, **20**(4), July 1998, pp. 724–767.

[WEB] The WebKit open source project. `http://www.webkit.org`.

[Wee06] Weeks, S. Whole program compilation in MLton. Invited talk at ML '06 Workshop, September 2006.

[WLZC11] Wang, Z., F. X. Lin, L. Zhong, and M. Chishtie. Why Are Web Browsers Slow on Smartphones? In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '11, Phoenix, Arizona, 2011. ACM.