2016 IEEE/ACM 38th IEEE International Conference on Software Engineering Companion

Engineering the Servo Web Browser Engine using Rust

Brian Anderson
Mozilla Research
banderson@mozilla.com

Lars Bergstrom
Mozilla Research
larsberg@mozilla.com

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

Josh Matthews
Mozilla
jdm@mozilla.com

Keegan McAllister*
mcallister.keegan@gmail.com

Jack Moffitt
Mozilla Research
jack@mozilla.com

Simon Sapin
Mozilla Research
ssapin@mozilla.com

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. Unfortunately, this language is complex (especially to new contributors!), challenging to write correct parallel code in, and highly susceptible to memory safety issues that potentially lead to security holes.

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 statically prevents many memory safety issues and provides direct support for parallelism and concurrency.

In this paper, we show how a language with an advanced type system can address many of the most common security issues and software engineering challenges in other browser engines, while still producing code that has the same performance and memory profile. This language is also quite accessible to new open source contributors and employees, even those without a background in C++ or systems programming. We also outline several pitfalls encountered along the way and describe some potential areas for future improvement.

Categories and Subject Descriptors

H.5.4 [**Information Interfaces and Presentation**]: Hypertext/Hypermedia; D.2.11 [**Software Engineering**]: Software Architectures—*languages, patterns*

Keywords

browser engine, Rust, Servo, concurrency, parallelism

*Work performed while employed at Mozilla

ICSE '16 Companion, May 14 - 22, 2016, Austin, TX, USA

ISBN 978-1-4503-4205-6/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2889160.2889229

1. INTRODUCTION

When most web browsers were originally designed, web pages were mostly static. Modern web browsers do not just display static pages, but run web applications with complexity similar to native software. From application suites such as Google Apps[1] to games based on the Unreal Engine,[2] modern browsers are a delivery platform for the types of rich media experiences historically tied to single hardware and operating system platforms. This shift has greatly increased the amount and complexity of code in a modern web engine as well as users' expectations around performance and security.

The heart of a modern web browser is its engine, 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. The use of C++ has enabled all of these browsers to achieve excellent sequential performance on a single web page, particularly on desktop computers. But, they all face several challenges:

- On mobile devices with lower processor speed but many more processors, these browsers do not provide the same level of interactivity [MTK+12, CFMO+13].
- In Gecko, roughly 50% of the security critical bugs are memory use after free, array out of range access, or related to integer overflow, all mistakes commonly made by even experienced C++ programmers with access to the best static analysis tools available.

[1] https://apps.google.com
[2] https://www.unrealengine.com/

- As the web has become more interactive, the mostly-sequential architecture of these engines has made it challenging to incorporate new features without sacrificing interactivity.

- With the growth in the popularity of other languages at the expense of C++, the number of volunteer contributors to the core C++ parts of these browser engine open source codebases has not grown apace with the increase in the size of the codebase.

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]. In Rust, errors such as off-by-one array access or memory buffer use after free are prevented by the language and its builtin libraries.

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.

With an average of 5 new contributors per week and several volunteers who have turned into key members of the project, we believe that Rust has helped Servo to lower the barrier to entry in systems programming.

Servo is currently over 800k lines of Rust code and implements enough of the web platform 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. 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 experimenting with the new programming language, Rust. In this experience report, we discuss the design and architecture of a modern web browser engine, show how using the Rust programming language has helped us to address the engineering challenges we have encountered when building the browser engine, and also touch on open problems and areas of future investigation.

2. BROWSERS

The architecture of all browsers are broadly the same, as demanded be the specifications of the web platform and the shared history and wisdom of browsers and their authors. As such, the steps Servo uses in Figure 1 to process a web page are similar to those 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 an implementation standpoint, there are two 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, 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. This *flow tree*, as it is named in Servo, describes the layout of DOM elements on the page, and may contain many more flows than previously existed in the DOM. For example, when a list item is styled to have an associated bullet, that bullet will itself be represented in the flow tree, though it is not part of the DOM.

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. These layers are then used to optimize interactive transformations, such as scrolling and certain animations, by only redrawing the buffers that have changed and otherwise simply recomposing the surface that has already been rendered into.

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] 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

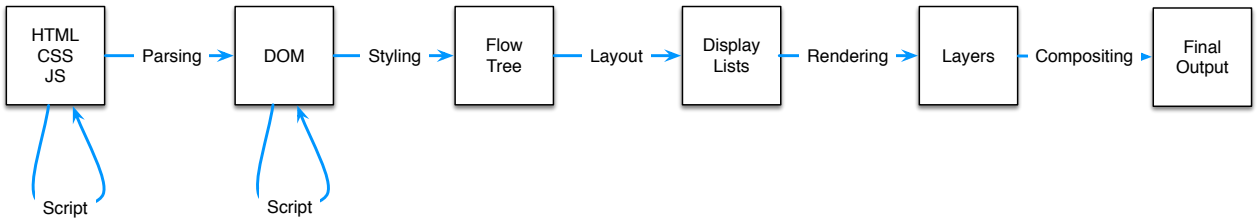

Figure 1: Processing stages and intermediate representations in a browser engine.

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 gives 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 **`unsafe`** blocks. Specifically, Rust prevents all of the following conditions:

- Dangling pointers
- Data races
- Integer overflow (in debug builds)
- Buffer overflow
- Iterator invalidation

Only integer overflow checking and prevention of buffer overflow require run-time checks, and buffer overflow checks are minimized idiomatically through the use of iterators.

Beyond performance and safety Rust features the expressive facilities of modern high level languages such as generics, algebraic data types, pattern matching, and closures. Abstractions in Rust are designed to have predictable and minimal performance impact, approaching the ideal of 'zero-cost' abstractions.

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

3.1 Feature overview

Rust's syntax draws heavily from C++, and many of its features will be familiar to modern programmers, though the details of both often differ in significant and interesting ways. What follows is a brief primer of the basics needed to understand the examples in this paper.

Local variables, declared with **`let`**, are immutable by default. To mutate a variable in Rust it must be declared **`mut`**. Both scalars and aggregates like structs are value types, allocated on the stack. The heap is accessed through library container types, the simplest of which are `Box`, a pointer to a value on the heap, and `Vec`, a dynamically-sized array of values. Both mutability and simple heap types are illustrated in Figure 2.

Rust's most prominent influence from the ML languages are algebraic data types, along with *pattern matching* to destructure them into their components. Rust calls algebraic data types *enums*, and they are the union of multiple types, each of which contains their own fields. Although superficially similar to C unions, they are significantly different in that instances of enums cannot be indiscriminately cast between variants. To access the values in an enum one must employ the **`match`** expression to first check the variant,

```
let max = 100;
let mut counter = 0;
let mut boxes: Vec<Box<i32>> = Vec::new();
while counter < max {
  // Append a boxed integer to a vector
  boxes.push(Box::new(counter));
  counter += 1;
}
```

Figure 2: Mutability and heap allocation.

```
enum Canvas2dMsg {
  BeginPath,
  ArcTo(Point2D<f32>, Point2D<f32>, f32),
  EndPath
}

let msg = Canvas2dMsg::BeginPath;

match msg {
  BeginPath => { }
  EndPath => { }
  ArcTo(point_a, point_b, radius) => {
    draw_arc_to(point_a, point_b, radius);
  }
}
```

Figure 3: Pattern matching to access the fields of an enum variant.

then bind references to its interior fields. In Figure 3 the values of a drawing message are extracted from an enum for further processing, with the bindings to the enum's fields established to the left of the fat arrow (=>) and the block of code operating on those bindings to the right.

Both enums and structs (which are substantially similar to C structs) may have associated instance methods and static methods, called with the dot (`.`) operator and the double-colon (`::`) operator, respectively. Static methods are common in Rust as they are used to construct values, conventionally with a method called `new`, as in `Point2D::new(0.0, 0.0)`. Notably, methods in Rust are dispatched statically, making them eligible for inlining and aggressive optimization. Methods are never dispatched dynamically through a function pointer in Rust unless explicitly requested.

Like many modern languages, Rust has first-class closures, anonymous functions that capture their environment. These use a compact notation with the arguments between pipes, and the body

```
// The single value we'll work with.
let val = Box::foo(0);

// Move the value in `val` to a new location
let moved_val = val;
// `val` is no longer accessible.
// The following will not compile.
// println(``{}'', val);

// Borrow and share the value immutably
// several times. This is done in a new
// block to limit the scope of the borrows.
{
  let shared1 = &moved_val;
  let shared2 = &moved_val;
  println!(``{}'', shared1);
  println!(``{}'', shared2);
  // Borrows are released at the end
  // of the block.
}
```

Figure 4: Ownership and borrowing.

between brackets: `|a, b| { a + b }`. The closure that takes no arguments and performs no computation is thus signified by the adorable series of characters, `||{}`. Rust closures are translated as efficiently as other types in Rust and can be inlined and optimized as well as any function.

Finally, Rust has a hygienic macro system, much more powerful than the C preprocessor. Macro invocations look like function invocations where the function name is appended with a bang (`!`). They are not functions, but are instead expanded in place at compile time, and perform syntactic transformations unavailable to functions. The common method of performing console output, `println!("{}", foo)`, is a macro. See Figure 5 for an example of closures and macros.

3.2 Ownership

The features of Rust described so far are common to many languages. Rust's novelty is in *ownership and borrowing*. In Rust, each value is uniquely owned by a single variable, and assigning that value to another variable transfers ownership of, and access to, that value to the receiver. Rust calls this transfer of ownership a *move*. When a value is allowed to go out of scope it is destroyed. In this way ownership-based resource management is like RAII[6] in C++.

To make using variables ergonomic, Rust can create temporary references to values with the ampersand (`&`) operator. References are simply pointers with limited scope, and may be either immutable or mutable. They are the only type of pointer in safe Rust. *The rules for borrowing ensure that no two pointers are ever mutably aliased, and that invalid memory is never accessible. This is what makes Rust memory safe.* See the example in Figure 4.

Ownership-based type systems are a kind of *affine* type system. Rust's model is influenced by the ownership model of Singularity OS [HLA+05], as well as the region systems in the Cyclone language [GMJ+02] and MLKit [TB98].

3.3 Ownership and concurrency

[6]"resource acquisition is initialization"

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

  // The `move` keyword here moves ownership of
  // the environment into the closure.
  thread::spawn(move || {
    *data = *data + 1;
  });
  // error: accessing moved value
  print!("{}", data);
}
```

Figure 5: Code that will not compile because it attempts to access mutable state from two threads.

```
fn main() {
  // An immutable shared pointer
  // (atomically reference counted)
  let data = Arc::new(1);
  let shared_data = data.clone();

  thread::spawn(move || {
    println!("{}", *shared_data);
  });
  print!("{}", *data);
}
```

Figure 6: Safely reading immutable state from two threads.

Because the Rust type system provides very strong guarantees about memory aliasing, Rust code is memory safe even in concurrent and multithreaded environments, and, perhaps surprisingly, is even guaranteed to be *data-race free*.

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 5 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 6 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 in Figure 7.

With relatively few simple rules, ownership in Rust enables foolproof task parallelism, but also data parallelism, by partitioning vectors and lending mutable references across 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 com-

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

  thread::spawn(move || {
    *shared_data.lock().unwrap() = 1;
  });

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

Figure 7: Safely mutating state from two threads.

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.

parison 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. While these measurements only reflect layout performance, they show that Servo is already faster than Gecko with even a single thread, and achieves speedups with multiple threads.

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 have an in-memory storage cost associated with the virtual function tables (sometimes many thousands of bytes[7]) 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, 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 so that library consumers can instantiate that AST directly instead of re-parsing the C++ header files for the templates.

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 thousands of modules from over 200 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.

Rust also requires that all memory is initialized. Failure to initialize memory also has led to crashes in Firefox.[8]

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 allocator 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.[9]

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

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

[8] https://bugzilla.mozilla.org/show_bug.cgi?id=1088731

[9] https://blog.mozilla.org/research/2014/08/26/javascript-servos-only-garbage-collector/

Language	Lines of Code
C or C++	1,187,939
Rust	816,158
HTML or JavaScript	248,768

Table 2: Lines of code in Servo (October 2015)

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. The ability to use C code has allowed us to smoothly integrate with many browser libraries, which has been critical for bootstrapping a browser without rewriting all of the lower-level libraries immediately, such as graphics rendering code, the JavaScript engine, font selection code, etc. Additionally, interoperation with C is required for components of a browser engine such as media decoders for DRM content, which are sometimes delivered by vendors as a binary along with a C API. 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.

In the future, we hope to expose our Rust libraries written for Servo to C, in order to reuse them in the Firefox web browser. A first potential library is our Rust-based URL parser.[10] While URL parsing seems like a simple and secure task, an audit of the public security bugs[11] shows that all four of the ones that come from URL parsing are related to either indexing out of range of a raw memory buffer or using data that had already been freed — both of which are prevented by Rust code.

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.

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. Much like other modern browsers, Servo contains many of its own specialized implementations of library functions that are tuned for the specific use cases of web browsers. This exhibits the control that Rust provides the programmer – high level abstractions are available, but should the need arise for a specialized abstraction, it can be written with ease by using lower-level constructs. 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.

[10]https://github.com/servo/rust-url

[11]Firefox public security bugs: http://mzl.la/1GkOiIC

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

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).

Many other systems programs have struggled with the complexity that arises in systems that have many concurrent threads. This complexity comes in the form of difficulty reasoning about whether a protocol will terminate, whether a message will eventually be handled, etc. Fortunately, Rust has a library that implements *session types*, which allow expressing a concurrent communication protocol in the type system and having the compiler enforce it [JML15]. We are just starting to use this library in Servo to manage the complexity of concurrency.

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 8, are written in a macro-based domain specific language that closely matches the format of the HTML specification.[12] Another 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 ca-

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

pabilities 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,[13] 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[14] 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.[15]

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.

4.8 Integer overflow

While more rare than memory safety bugs, overflowing an integer is still a source of some bugs in modern systems software. In Rust, integer overflow is checked by default in debug builds, and we have caught several potential bugs simply by compiling the code and running our tests with debugging enabled in our automation systems.

4.9 New contributors

As mentioned in Section 1, Servo gains roughly 5 new contributors per week. While part of that is due to tremendous outreach efforts on the part of the team, we believe that the accessibility of Rust to non-systems programmers plays a large role as well. Almost none of our new contributors come to Servo with C++ experience, and nearly a third of our full-time staff members did not program in it professionally before beginning work on Servo.

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

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

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

4.10 Modular development

Most large systems projects have a *monorepo* design for their source code control (SCC) and build system. That is, all of the millions of lines of code that make them up live in a single SCC system and use a toplevel build system that drives it all in one step. This organization is great for developers on the system, as they can make coordinated changes to many parts of the system and complete them in one step. However, it has meant that many parts of the browser engine, such as the HTML parser, CSS selector matching, or URL parsing, could not easily be separated out and reused by other projects, so there are separate projects (which do not typically implement quite the same standards as the implementations in browsers!) that attempt to provide these facilities to other programs.

In Servo, we instead use a *polyrepo* design, which is inherited from Rust's default tooling. Our source code is stored across many repositories on GitHub[16], and we use Rust's Cargo[17] build system to compile, link, and manage the dependencies between all of the parts that make up Servo. While this design has the downside that coordinated changes spanning multiple parts of the system take many steps, it has the advantage that many of Servo's submodules are used and contributed to by developers who do not directly work on Servo.

5. OPEN PROBLEMS

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

5.1 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.

5.2 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.

5.3 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 mutation 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

[16]http://www.github.com

[17]http://crates.io

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 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.

7. CONCLUSION

In this experience report, we have described our experiences using the new Rust programming language to develop Servo. We cannot yet quantify the amount of additional work that is required by the stricter typesystem to author Rust code instead of C or C++. But, we have found that the Rust language is accessible to new developers who only have experience in C++, Python, or JavaScript — even for writing core performance- and memory-sensitive libraries. Further, with a relatively small team and short time, we have written a browser engine that is increasingly competitive both in performance and functionality with other modern browser engines.

The strongest evidence that Rust is not too expensive to write, though, is that it can statically verify the absence of memory safety errors. Other systems projects — including all currently shipping browser engines – have spent great amounts of time writing and reviewing their C or C++ code, written fuzzers, and used internal and external static analysis tools, but their security vulnerabilities are still primarily memory safety errors. For projects where memory safety errors need to be eliminated, Rust is a superior choice.

8. 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.

We would like to thank Robert Clipsham, Valentin Gosu, David Herman, Tim Kuehn, Lindsey Kuper, Ms2ger, Ben Striegel, and previous anonymous reviewers for their comments on drafts of this paper.

9. ADDITIONAL AUTHORS

10. 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.

[JML15] Jespersen, T. B. L., P. Munksgaard, and K. F. Larsen. Session types for rust. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*, WGP 2015, Vancouver, BC, Canada, 2015. ACM, pp. 13–22.

[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.