Quantum-Servo : Engineering Parallel Web Browser Engine

CS 17L4 Seminar

CSU 142 46 12150858 Sajith A Rahim

B. Tech. Computer Science & Engineering

Department of Computer Engineering
Model Engineering College Thrikkakara
Kochi 682021
Phone: +91.484.2575370
http://www.mec.ac.in hodcs@mec.ac.in

2017

Model Engineering College Thrikkakara
Dept. of Computer Engineering

C E R T I F I C A T E

This is to certify that, this report titled **Quantum-Servo : Engineering Parallel Web Browser Engine** is a bonafide record of the **CS 17L4 Seminar** presented by

CSU 142 46 **12150858** **Sajith A Rahim**

Seventh Semester B.Tech. Computer Science & Engineering

scolar, under our guidance and supervision, in partial fulfillment of the requirements for the award of the degree, **B. Tech. Computer Science and Engineering** of **Cochin University of Science & Technology.**

Guide	Coordinator
Divya K.B	Remyakrishnan P
Asst. Professor	Asst. Professor
Computer Engineering	Computer Engineering

Head of the Department

September 15, 2021

Manilal D L
Associate Professor
Computer Engineering

Acknowledgement

I would like to express my sincere gratitude to everyone who assisted me in conducting this Seminar. My heartfelt gratitude to the coordinator Assistant Remyakrishnan P, Department of Computer Engineering, for giving me permission to commence this seminar. I express my gratitude to Principal Prof. (Dr.) V. P. Devassia and Associate Prof. Manilal D L, HOD of the Department of Computer Engineering, Model Engineering College. Special thanks to my guide Divya KB, Department of Computer Engineering, whose guidance and encouragement helped me through out this endeavour.

Sajith A Rahim

Abstract

All modern web browsers Internet Explorer, Firefox, Chrome, Opera, and Safari have a core rendering engine written in C++. Almost all of these browsers depend on engines developed prior to the 21 century and is quite large in size (6 million lines of code).The design and flow of these engines are based on the standards that came into existence in its timeline -improvising and adding to the existing engine .Hence they contain support for now deprecated standards and web elements that existed before adding to the bulkiness of the engine. Optimization of performance and refactoring the process pipeline is hard and way complex to implement on these engines.

Mozilla Research is developing Servo, a parallel web browser engine, to exploit the benefits of parallelism and concurrency in the web rendering pipeline. Parallelization results in improved performance depending on the workload, the workload of a browser is dependent on the web page it is rendering. In late 1990s C/C++ was selected as the language of choice for writing Browser Engines 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 challenging to write correct parallel code in, and highly susceptible to memory safety issues that potentially lead to security holes. Servo aims 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 (developed by Mozilla for this purpose specifically), 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.

Quantum Project is an extension of Servo project aimed to build the next-generation web engine for Firefox users, building on the Gecko engine as a solid foundation. Quantum will leverage the fearless concurrency of Rust and high-performance components of Servo to bring more parallelization and GPU offloading to Firefox.

In this report, I aim to explain the components and working of Servo based components forked to Quantum Browser Engine and explain what advantages it brings to us in terms of user experience, security and performance.

Contents

List of Figures

Chapter 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:

- Trident/Spartan, the engine in Internet Explorer [IE]
- Webkit[WEB]/Blink, the engine in Safari [SAF],Chrome [CHR], and Opera [OPE]
- 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.

1. 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].
2. 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.
3. 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.
4. ith 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.

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, they are using a new systems programming language,Rust [RUS]. For parallelism and power, they scale across a wide variety of hardware by building either data- or task-parallelism, as appropriate, into each part of the web platform.

Additionally, they 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 800k 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, Mozilla 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 report, I 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.

Chapter 2

A WEB BROWSER ENGINE

2.1 How does it Work?

A web browser is a piece of software that loads files (usually from a remote server) and displays them locally, allowing for user interaction.The industry term for that part is browser engine, and without one, you would just be reading code instead of actually seeing a website. Firefoxs browser engine is called Gecko.

Its pretty easy to see the browser engine as a single black box, sort of like a TV- data goes in, and the black box figures out what to display on the screen to represent that data. The question today is: How? What are the steps that turn data into the web pages we see?

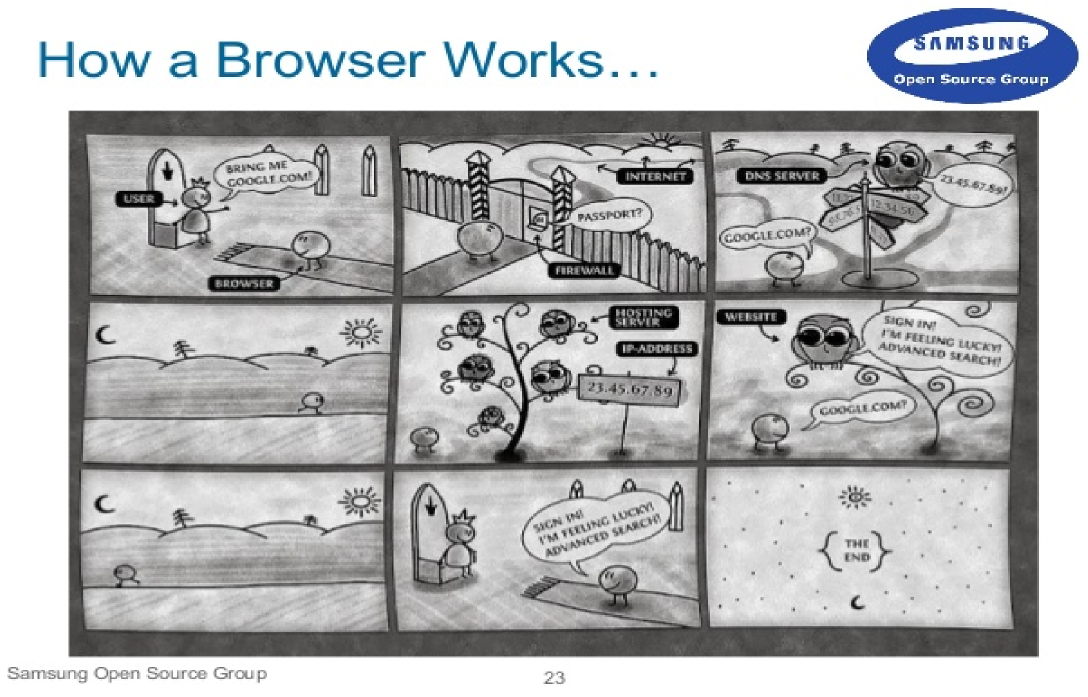


Figure 2.1: Graphical representation of Web Rendering Process

The data that makes up a web page is lots of things, but its mostly broken down into 3 parts:

- code that represents the structure of a web page
- code that provides style: the visual appearance of the structure
- code that acts as a script of actions for the browser to take: computing, reacting to user actions, and modifying the structure and style beyond what was loaded initially

The browser engine combines structure and style together to draw the web page on your screen, and figure out which bits of it are interactive. It all starts with structure. When a browser is asked to load a website, its given an address. At this address is another computer which, when contacted, will send data back to the browser. The particulars of how that happens are a whole separate article in themselves, but at the end the browser has the data. This data is sent back in a format called HTML, and it describes the structure of the web page.

2.2 How does a browser understand HTML?

Browser engines contain special pieces of code called parsers that convert data from one format into another that the browser holds in its memory 1. The HTML parser takes the HTML, something like:

```
<section>
 <h1 class="main-title">Hello!</h1>
 <img src="http://example.com/image.png">
</section>
```

And parses it, understanding:
Okay, theres a section. Inside the section is a heading of level 1, which itself contains the text: Hello! Also inside the section is an image. I can find the image data at the location: http://example.com/image.png

The in-memory structure of the web page is called the Document Object Model, or DOM. As opposed to a long piece of text, the DOM represents a tree of elements of the final web page: the properties of the individual elements, and which elements are inside other elements

In addition to describing the structure of the page, the HTML also includes addresses where styles and scripts can be found. When the browser finds these, it contacts those addresses and loads their data. That data is then fed to other parsers that specialize in those data formats. If scripts are found, they can modify the page structure and style before the file is finished being parsed. The style format, CSS, plays the next role in our browser engine.

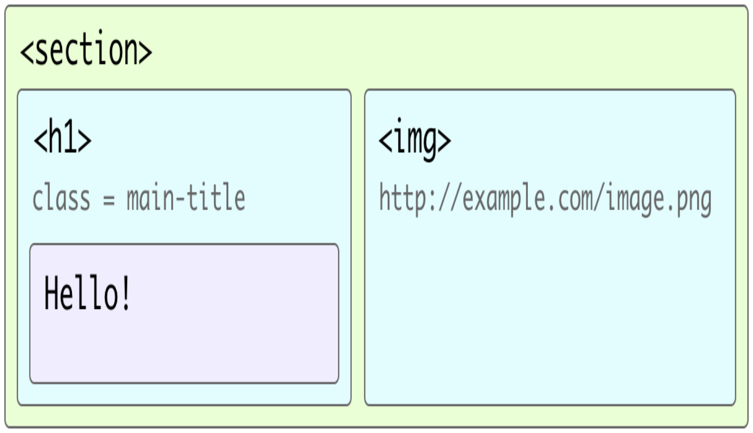


Figure 2.2: DOM

2.3 Rendering with Style Sheets

CSS is a programming language that lets developers describe the appearance of particular elements on a page. CSS stands for Cascading Style Sheets, so named because it allows for multiple sets of style instructions, where instructions can override earlier or more general instructions (called the cascade). A bit of CSS could look like the following:
CSS is largely broken up into groupings called rules, which themselves consist of two parts. The first part is selectors. Selectors describe the elements of the DOM (remember those from above?) being styled, and a list of declarations that specify the styles to be applied to elements that match the selector. The browser engine contains a subsystem called a style engine whose job it is to take the CSS code and apply it to the DOM that was created by the HTML parser.

For example, in the above CSS, we have a rule that targets the selector section, which will match any element in the DOM with that name. Style annotations are then made for each element in the DOM. Eventually each element in the DOM is finished being styled, and we call this state the computed style for that element. When multiple competing styles are applied to the same element, those which come later or are more specific wins. Think of stylesheets as layers of thin tracing paper- each layer can cover the previous layers, but also let them show through.

Once the browser engine has computed styles, its time to put it to use! The DOM and the computed styles are fed into a layout engine that takes into account the size of the window being drawn into. The layout engine uses various algorithms to take each element and draw a box that will hold its content and take into account all the styles applied to it.

When layout is complete, its time to turn the blueprint of the page into the part you see. This process is known as painting, and it is the final combination of all the previous steps. Every box that was defined by layout gets drawn, full of the content from the DOM and with styles from the CSS. The user now sees the page, reconstituted from the code that defines it.

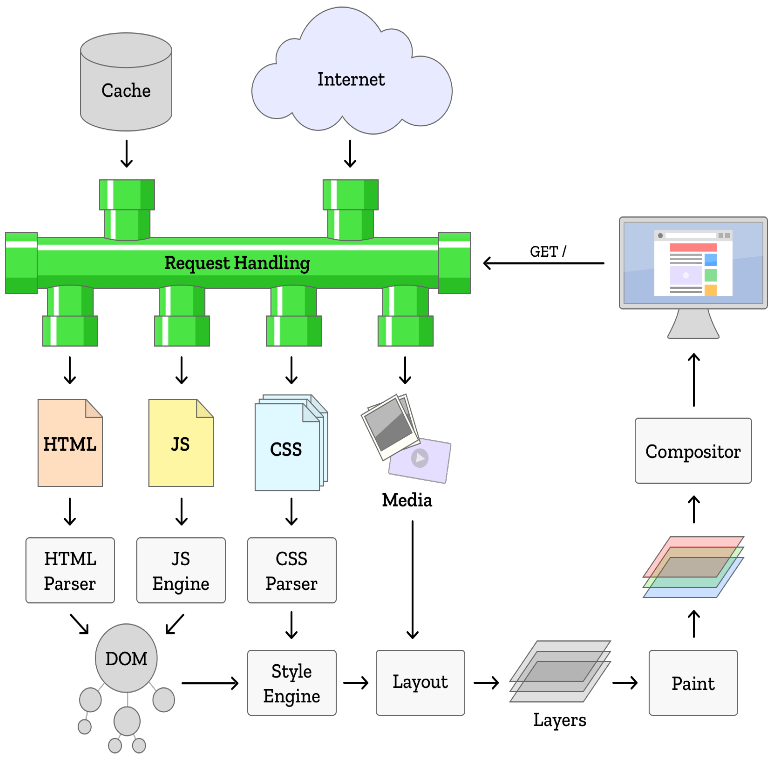


Figure 2.3: A complete Web Browser Engine

When the user scrolled the page, we would re-paint, to show the new parts of the page that were previously outside the window. It turns out, however, that users love to scroll! The browser engine can be fairly certain it will be asked to show content outside of the initial window it draws (called the viewport). More modern browsers take advantage of this fact and paint more of the web page than is visible initially. When the user scrolls, the parts of the page they want to see are already drawn and ready. As a result, scrolling can be faster and smoother. This technique is the basis of compositing, which is a term for techniques to reduce the amount of painting required.

Additionally, sometimes we need to redraw parts of the screen. Maybe the user is watching a video that plays at 60 frames per second. Or maybe theres a slideshow or animated list on the page. Browsers can detect that parts of the page will move or update, and instead of re-painting the whole page, they create a layer to hold that content. A page can be made of many layers that overlap one another. A layer can change position, scroll, transparency, or move behind or in front of other layers without having to re-paint anything! Pretty convenient.

Sometimes a script or an animation changes an elements style. When this occurs, the style engine need to re-compute the elements style (and potentially the style of many more elements on the page), recalculate the layout (do a reflow), and re-paint the page. This takes a lot of time as computer-speed things go, but so long as it only happens occasionally, the process wont negatively affect a users experience.

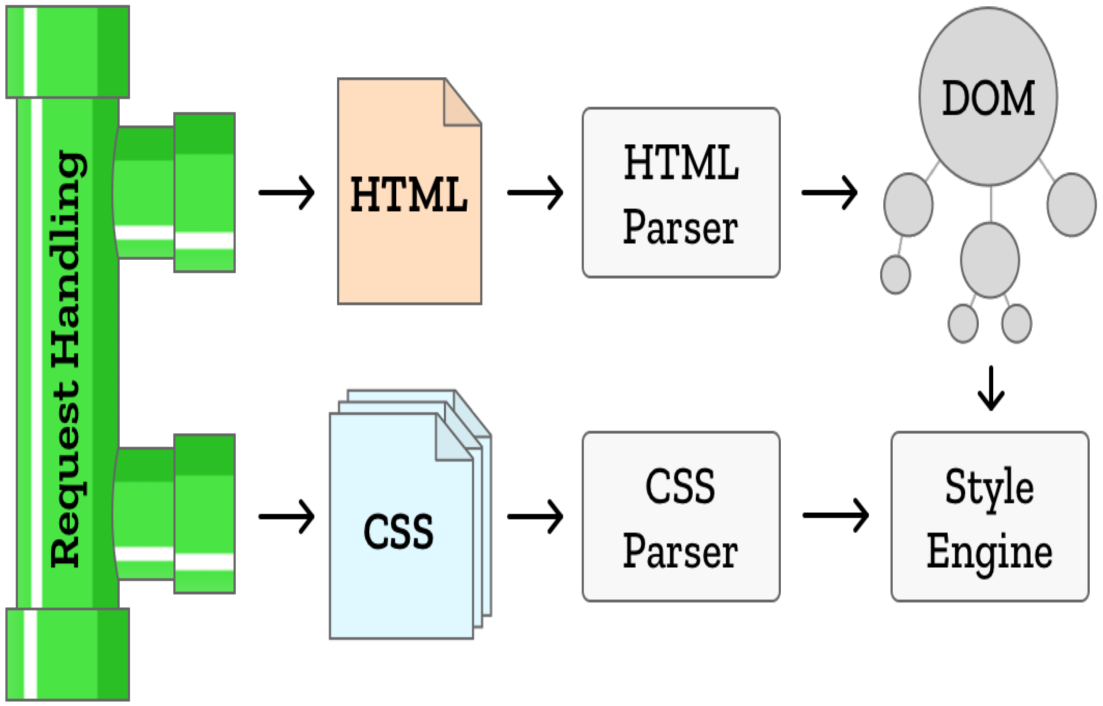


Figure 2.4: Web Engine wih CSS Parser

In modern web applications, the structure of the document itself is frequently changed by scripts. This can require the entire rendering process to start more-or-less from scratch, with HTML being parsed into DOM, style calculation, reflow, and paint.

Chapter 3

NEED FOR A NEW ENGINE

3.1 Motivations

Mozilla chose the Quantum name because the next-generation engine will provide a quantum leap in performance on mobile and desktop. To put this in perspective, Gecko started out in the Netscape browser released in 1997 and has been modified over time to support new technologies.

So why do we need a new browser engine ?

3.1.1 Existing Engines are really Huge

```
\$ hg clone
http://hg mozilla org/mozi I la-centra I \$ cd mozilla-central

\$ find . -name '*.c' -or -name '*.cpp' -or -name '* h' -or -name '*.tbl' | xargs wc -I
```

OUTPUT : 10127726 This only shows the C/C++ code while the major part of Mozilla Gecko is written in Javascript.

Gecko engine (Since 1997) has at present nearly 6 million lines of code .This makes it really hard to optimize the engine or refactor it.

3.1.2 Path Dependence.

Path dependence basically means history matters. First Firefox was designed back in 1993 ie more than 20 years ago. We had a nearly Utopian web back then like documents were documents,images were images there wasnt much Javascript back then.As a result there wasnt much security issues.

A lot of deprecated code got accumulated in the browser engine.There was a lot of standards that were proposed over the years many got implemented as base web standards and then later got removed. Some easy examples are marquee etc. Even though they got depricated and are not used but there is support for them still. Designers had to make architectural decisions based on these standards which are no longer relevant like Mozilla still supports Windows 95 Server until itll be deprecated in 5 more years or so.

3.1.3 No Concept of Parallel Processing

All the systems as well as design decisions made back then were based on the single processor design.Since the browser engine are made top on those designs there hasnt been much focus on parallel processing or threading.This is mainly because of the gruesome task of refactoring the entire code base inorder to achieve that.

3.1.4 C++ is depressing when it comes to Security and Parallel Processing

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 browser 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 .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. Most of the major security bugs in engines are directly related to C memory model.

So there rises a need to have a new language to code the parallel browser engine. A language which solves the shortcomings of C/C++ memory model. Mozillas answer to that is **RUST** Programming Language.

3.2 RUST

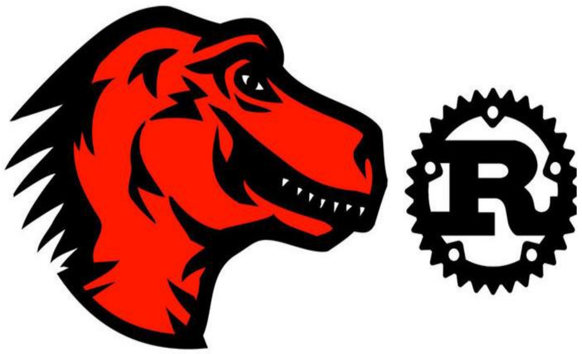


Figure 3.1: RUST Icon

3.2.1 Introduction

Since C++ is poorly suited to tackle the above problems, Servo is written in Rust, a new language designed specifically with Servo's requirements in mind. Rust provides a task-parallel infrastructure and a strong type system that enforces memory safety and data race freedom.

Rust is meant to be fast, safe, and reasonably easy to program in. Its also intended to be used widely, and not simply end up as a curiosity or an also-ran in the language sweepstakes. Good reasons abound for creating a language where safety sits on equal footing with speed and development power. After all, theres a tremendous amount of softwaresome of it driving critical infrastructurebuilt with languages where safety wasnt the first concern.

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.

3.2.2 What makes Rust a better Development Language

Rust started as a Mozilla research project partly meant to reimplement key components of the Firefox browser. A few key reasons drove that decision: Firefox deserved to make better use of modern, multicore processors; and the sheer ubiquity of web browsers means they need to be safe to use. But those benefits are needed by all software, not just browsers, which is why Rust evolved into a language project from a browser project. Rust accomplishes its safety, speed, and ease of use through the following characteristics:

Rust satisfies the need for speed.

Rust code compiles to native machine code across multiple platforms. Binaries are self-contained, with no runtime, and the generated code is meant to perform as well as comparable code written in C or C++.

Rust wont compile programs that attempt unsafe memory usage.

Most memory errors are discovered when a program is running. Rusts syntax and language metaphors ensure that common memory-related problems in other languagesnull or dangling pointers, data races, and so onnever make it into production. The compiler flags those issues and forces them to be fixed before the program ever runs.

Rust controls memory management via strict rules.

Rusts memory-management system is expressed in the languages syntax through a metaphor called ownership. Any given value in the language can be owned, or held/manipulated, only by a single variable at a time.

The way ownership is transferred between objects is strictly governed by the compiler, so there are no surprises at runtime in the form of memory-allocation errors. The ownership approach also means there is no garbage-collected memory management, as in languages like Go or C. (That also gives Rust another performance boost.) Every bit of memory in a Rust program is tracked and released automatically through the ownership metaphor.

Rust lets you live dangerously if you need to, to a point.

Rusts safeties can be partly suspended where you need to manipulate memory directly, such as dereferencing a raw pointer la C/C++. The key word is partly, because Rusts memory safety operations can never be completely disabled. Even then, you almost never have to take off the seatbelts for common use cases, so the end result is software thats safer by default.

Rust is designed to be easy to use.

None of Rusts safety and integrity features add up to much if they arent used. Thats why Rusts developers and community have tried to make the language as useful and welcoming as possible to newcomers.

Everything needed to produce Rust binaries comes in the same package. External compilers, like GCC, are needed only if you are compiling other components outside the Rust ecosystem (such as a C library that youre compiling from source). Microsoft Windows users are not second-class citizens, either; the Rust tool chain is as capable there as it is on Linux and MacOS.

On top of all that, Rust provides several other standard-issue items youd expect or want:

- Support for multiple architectures and platforms. Rust works on all three major platforms: Linux, Windows, and MacOS. Others are supported beyond those three. If you want to cross-compile, or produce binaries for a different architecture or platform than the one youre currently running, a little more work is involved, but one of Rusts general missions is to minimize the amount of heavy lifting needed for such work. Also, although Rust works on the majority of current platforms, its not its creators goal to have Rust compile absolutely everywherejust on whatever platforms are popular, and wherever they dont have to make unnecessary compromises to do so.

- Powerful language features. Few developers want to start work in a new language if they find it has fewer, or weaker, features than the ones theyre used to. Rusts native language features compare favorably to what languages like C++ have: Macros, generics, pattern matching, and composition (via traits) are all first-class citizens in Rust.

- A useful standard library. One part of Rusts larger mission is to encourage C and C++ developers to use Rust instead of those languages whenever possible. But C and C++ users expect to have a decent standard librarythey want to be able to use containers, collections, and iterators, perform string manipulations, manage processes and threading, perform network and file I/O, and so on. Rust does all that, and more, in its standard library. Because Rust is designed to be cross-platform, its standard library can contain only things that can be reliably ported across platforms. Platform-specific functions like Linuxs epoll have to be supported via functions in third-party libraries such as libc, mio, or tokio.

- Third-party libraries, or crates. One measure of a languages utility is how much can be done with it thanks to third parties. Cargo, the official repository for Rust libraries (called crates) lists some ten thousand crates. A healthy number of them are API bindings to common libraries or frameworks, so Rust can be used as a viable language option with those frameworks. However, the Rust community does not yet supply detailed curation or ranking of crates based on their overall quality and utility, so you cant easily tell what works well.

- IDE tools. Again, few developers want to embrace a language with little or no support in the IDE of their choice. Thats why Rust recently introduced the Rust Language Server, which provide live feedback from the Rust compiler into an IDE such as Microsoft Visual Studio Code.

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 controlover 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*

3.2.3 Syntax

Rust has struct and enum types (similar to Standard MLs record types and datatypes 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 bytes6) 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 toavoid 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 with their attempts to build a safe language that required minimal porting effort for even complicated C code.

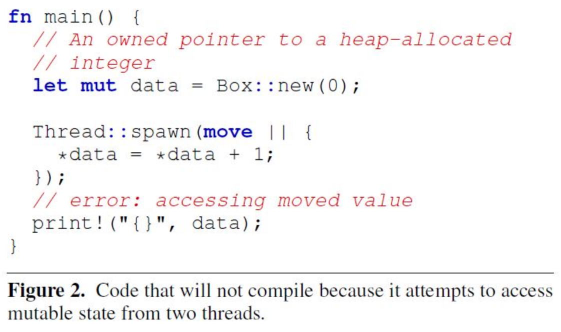
Rusts syntax draws heavily from C++, and many of its features will be familiar to modern programmers, though the details of both often differ in significant and interesting ways. What follows is a brief primer of the basics needed to understand the examples in this paper. Local variables, declared with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

Rusts 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,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 enums fields established to thenleft 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 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.

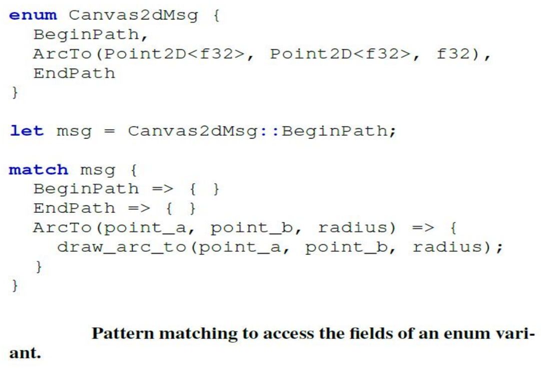


```
fn main() {
  // An owned pointer to a heap-allocated
  // integer
  let mut data = Box::new(0);

  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.

Figure 3.2: Code trying to access mutable state simultaneously (Won't Compile)



```
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);
  }
}
```

Pattern matching to access the fields of an enum variant.

Figure 3.3: Pattern Matching Across Enum

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.

3.2.4 Memory management

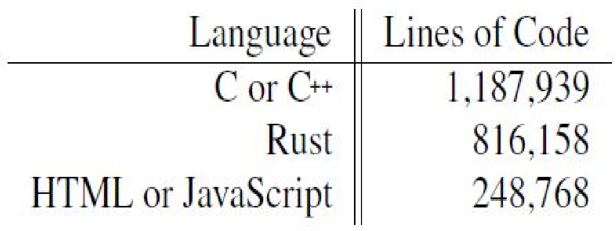
Rust has an affine type system that ensures every value is used at most once. One result of this fact is that in the more than two years since Servo has been under development, we have encountered zero use-after-free memory bugs in safe Rust code. Given that these bugs make up such a large portion of 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.

3.2.5 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.



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

Figure 3.4: Lines Of Code in Servo (As Of Nov 2015)

3.2.6 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 de fault 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,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.

3.2.7 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 shownin 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$_{c}har!macrowillcauseanearlyreturnfromthefunctionthatconte$

3.2.8 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.

Chapter 4

SERVO THE PARALLEL BROWSER ENGINE

4.1 Overview

Servo is an experimental web browser layout engine being developed by Mozilla Research, with Samsung porting it to Android and ARM processors. The prototype seeks to create a highly parallel environment, in which many components (such as rendering, layout, HTML parsing, image decoding, etc.) are handled by fine-grained, isolated tasks. Source code for the project is written in the Rust programming language.

Two significant components used by Servo are based on pre-existing C++ code from Mozilla. JavaScript support is provided by SpiderMonkey, and the 2D graphics library Azure is used to interface with OpenGL and Direct3D.

Servo makes use of GPU acceleration to render web pages more quickly. Servo is significantly faster, in certain benchmarks, than Gecko, Mozilla's other layout and rendering engine, as of November 2014.

4.2 Development Timeline.

Mozilla Research's projects diagram featuring Servo and Development of Servo began in 2013. The very first commit on 8 February 2012 did not contain any source code. The first rudimentary code commit occurred on 27 March 2012.

On 3 April 2013 Mozilla announced that they and Samsung collaborate on Servo. As of the 30th of June 2016, a preview version is available for download. This is marked as 0.0.1 and is available for Mac and Linux.As of the 13th of April 2017, builds are now available for Windows as well.

Servo is focused both on supporting a full Web browser, through the use of the purely HTML-based user interface Browser.html and on creating a solid, embeddable engine. Although Servo was originally a research project, it was implemented with the goal of having production-quality code and is in the process of shipping several of its components in the Firefox browser.

Figure 4.1: Servo Logo

4.3 Components

The Quantum project is composed of several sub-projects.

4.3.1 CSS

Servo's parallel style sheet system integrated into Gecko. Benchmarks suggest that performance scales linearly with number of CPU cores.

4.3.2 Render

Servo's rendering architecture, called Webrender, integrated into Gecko. Webrender replaces the immediate mode drawing model with a retained mode model that is more easily accelerated by the GPU by taking advantage of CSS/DOM's similarity to a scene graph. Worst-case scenario rendering in testing exceeds 60 frames per second.

4.3.3 Compositor

Gecko's existing compositor moved to its own process, isolating browser tabs from graphics driver related crashes. Since compositor crashes will not bring down the browser content process, the compositor process can be restarted transparently without losing user data. Shipped to users in Firefox 53.

4.3.4 DOM

Loosely inspired by Servo's Constellation architecture and Opera's Presto engine, Quantum DOM uses cooperatively scheduled threads within the DOM to increase responsiveness without increasing the number of processes and, thus, memory usage.

4.3.5 Flow

An umbrella for user visible performance improvements driven by a team that works across Gecko components. Currently focused on real user performance improvements on major webapps, primarily the G Suite and Facebook

4.3.6 Photon

A UI refresh of the entire application, with a strong focus on improving UI performance. Treated as a sister project to Quantum Flow

4.3.7 Network

Improve the performance of Necko, Gecko's networking layer, by moving more network activity off the main thread, context dependent prioritization of networking streams, and racing the cache layer with the network.

4.4 Performance Improvements

Before going into explaining what the performance details of Servo over other browser engine lets describe the standards for the measure of performance.

Performance of browser engine is usually Quantized based on
1. Page Load Time
2. JS Execution time
3. Responsiveness - Parallelism
4. Power Usage

Out of the above the first two has been the prime focus of Web Engineers for the past few decades.There has been much development on those two so the prime focus in building servo was the optimization of the latter two.

4.5 Parrallelization and Concurrency

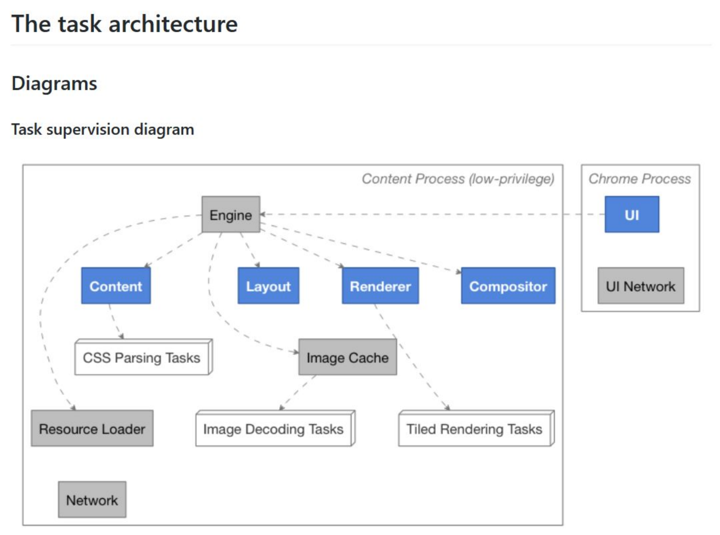
Concurrency is the separation of tasks to provide interleaved execution. Parallelism is the simultaneous execution of multiple pieces of work in order to increase speed.

4.5.1 Features

Here are some ways that we take advantage of both:

Task-based architecture

Major components in the system should be factored into actors with isolated heaps, with clear boundaries for failure and recovery. This will also encourage loose coupling throughout the system, enabling us to replace components for the purposes of experimentation and research.


The task architecture

Diagrams

Task supervision diagram

Figure 4.2: Task Architecture

Concurrent rendering.

Both rendering and compositing are separate threads, decoupled from layout in order to maintain responsiveness. The compositor thread manages its memory manually to avoid garbage collection pauses.

Tiled rendering.

We divide the screen into a grid of tiles and render each one in parallel. Tiling is needed for mobile performance regardless of its benefits for parallelism.

Layered rendering.

We divide the display list into subtrees whose contents can be retained on the GPU and render them in parallel.

Selector matching.

This is an embarrassingly parallel problem. Unlike Gecko, Servo does selector matching in a separate pass from flow tree construction so that it is more easily parallelized.

Parallel layout.

We build the flow tree using a parallel traversal of the DOM that respects the sequential dependencies generated by elements such as floats.

Text shaping.

A crucial part of inline layout, text shaping is fairly costly and has potential for parallelism across text runs. Not implemented.

Parsing.

We have written a new HTML parser in Rust, focused on both safety and compliance with the specification. We have not yet added speculation or parallelism to the parsing.

Decoding of other resources.

This is probably less important than image decoding, but anything that needs to be loaded by a page can be done in parallel, e.g. parsing entire style sheets or decoding videos.

GC JS concurrent with layout.

Under most any design with concurrent JS and layout, JS is going to be waiting to query layout sometimes, perhaps often. This will be the most opportune time to run the GC.

In Servo's rendering architecture, each tab is a "constellation" that gets assigned a number of rendering pipeline threads. The tab's contents are broken down into a DOM tree, then some number of pipelines are started and begin running a work-stealing algorithm to process the nodes in the tree. Each of the pipelines has its own renderer, script engine, and layout engine, although they must (by necessity) share some items that are reused in the page.

Parrallelism in Web can be classified as :

- Parallel layout
- Parallel styling

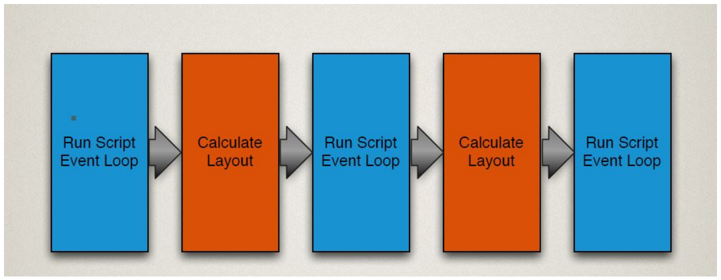


Figure 4.3: Single Threaded Browser Execution

As shown almost all browsers at present are single-threaded.So the control has to wait around till the frames have executed before moving on to the next.

Servos layout operates on a flow tree, which is similar to the render tree in WebKit or Blink and the frame tree in Gecko. We call it a flow tree rather than a render tree because

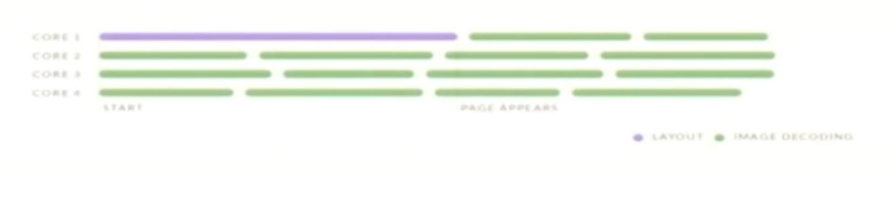


Figure 4.4: Traditional Layout Processing

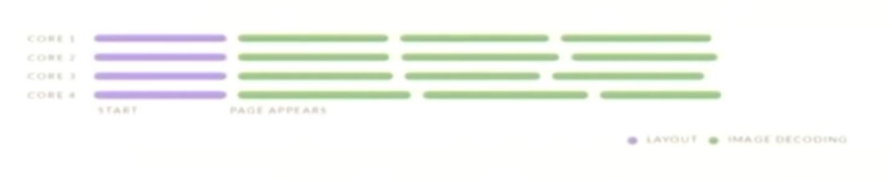


Figure 4.5: Parrallel Layout Processing

it consists of two separate data types: flows, which are organized in a tree, and boxes, which belong to flows and are organized in a flat list. Roughly speaking, a flow is an object that can be laid out in parallel with other flows, while a box is a box that must be laid out sequentially with other boxes in the same flow. If youre familiar with WebKit, you can think of a box as a RenderObject, and if youre familiar with Gecko, you can think of a box as a nsFrame. We want to lay out boxes in parallel as much as possible in Servo, so we group boxes into flows that can be laid out in parallel with one another.

In Blink, layout will create a RenderTree with RenderObjects that have geometric information and know how to paint themselves.

In Servo, layout will generate a FlowTree and later DisplayLists are created from it.

So the goal is to run layout and script parallelly thus speeding up the rendering process.

Solution to this is The *Copy-on-Write DOM*

Servo's DOM is a tree with versioned nodes that may be shared between a single writer and many readers. The DOM uses a copy-on-write strategy to allow the writer to modify the DOM in parallel with readers. The writer is always the content task and the readers are always layout tasks or subtasks thereof.

DOM nodes are Rust values whose lifetimes are managed by the JavaScript garbage collector. JavaScript accesses DOM nodes directlythere is no XPCOM or similar infrastructure.

The interface to the DOM is not currently type safeit is possible to manage nodes incorrectly and end up dereferencing bogus pointers. Eliminating this unsafety is a high-priority, and necessary, goal for the project; as DOM nodes have a complex life cycle this will present some challenges.

C.O.W. DOM SAFETY

- Rust type system enforces safety
- Layout and script tasks see different types for the DOM tree and both types have distinct implementations
- Leaks are not allowed; nodes can't escape

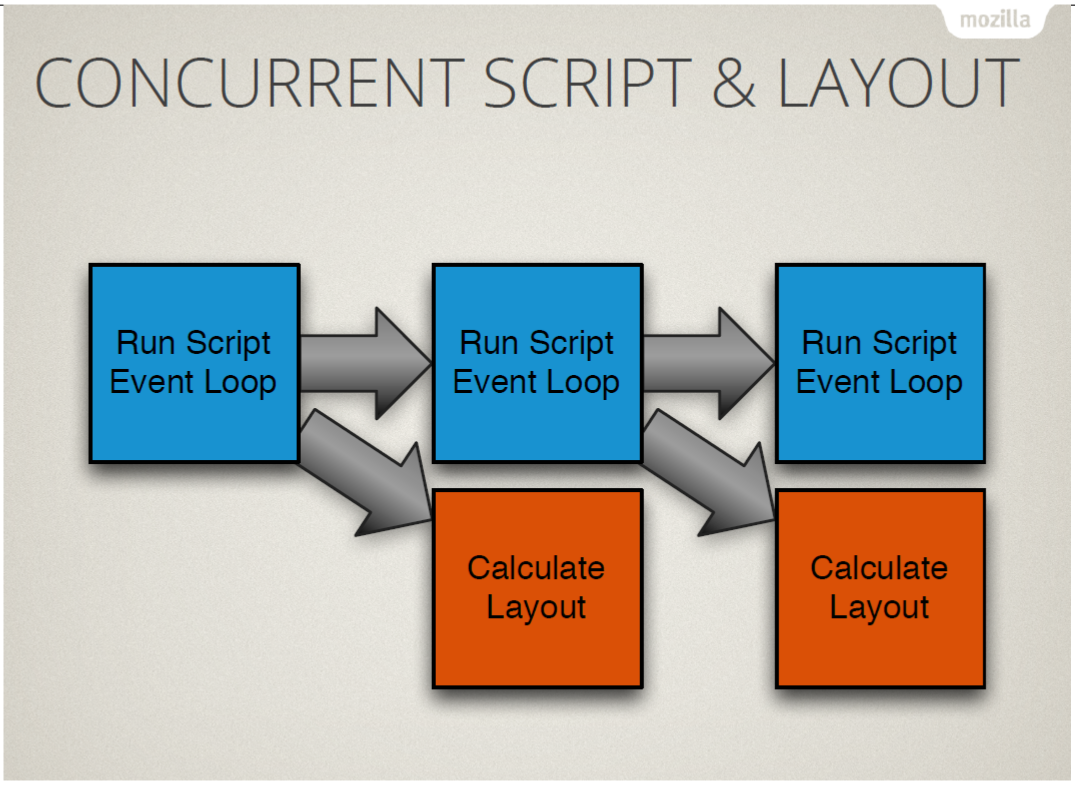
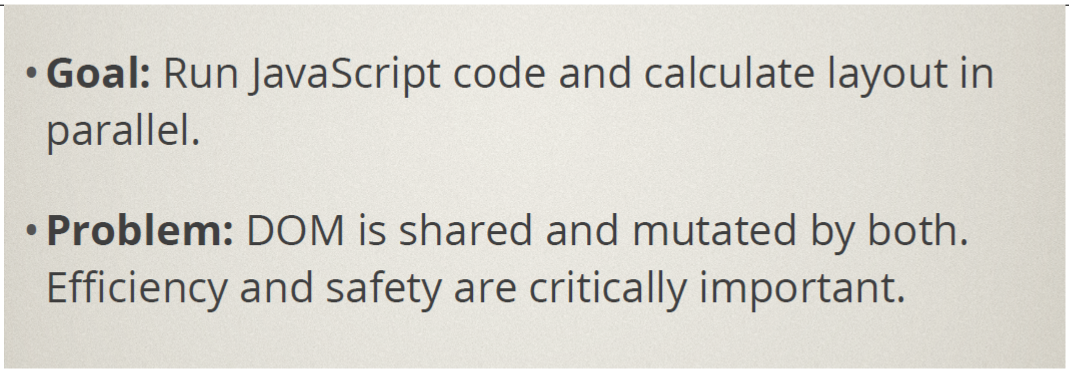


Figure 4.6: Concurrent Script & Layout

4.5.2 Challenges

Parallel-hostile libraries. Some third-party libraries we need don't play well in multi-threaded environments. Fonts in particular have been difficult. Even if libraries are technically thread-safe, often thread safety is achieved through a library-wide mutex lock, harming our opportunities for parallelism.

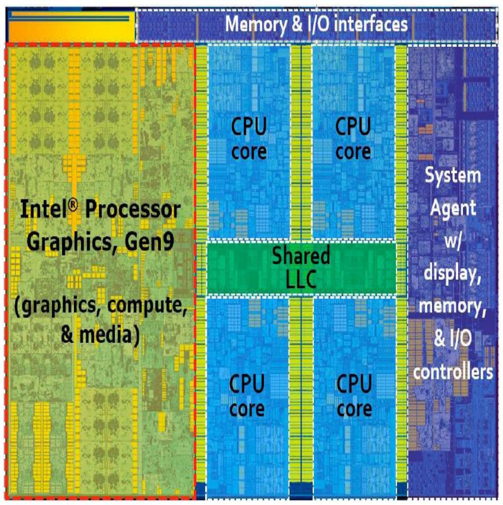
Too many threads. If we throw maximum parallelism and concurrency at everything, we will end up overwhelming the system with too many threads.



- **Goal:** Run JavaScript code and calculate layout in parallel.
- **Problem:** DOM is shared and mutated by both. Efficiency and safety are critically important.

4.6 WebRender

WebRender Motivation


Memory & I/O interfaces
Intel® Processor Graphics, Gen9 (graphics, compute, & media)
CPU core
CPU core
Shared LLC
CPU core
CPU core
System Agent w/ display, memory, & I/O controllers

If we take a look at the processor we can see that half the chip is GPU.So instead of focusing on CPU Cores alone why not use GPU also.WebRender is the result of an attempt to achieve this.

Webrender is an experimental renderer for Servo that aims to draw web content like a modern game engine.

4.6.1 What does this mean?

1. Specialized renderer for web content.
    - Not a general purpose vector graphics API.
    - Complex items such as canvas can be rendered on CPU.

- Make the common case fast.

2. Use GPU to do all rasterization where possible : Items not well suited to GPU rasterizing can be drawn on the CPU and cached in textures.

3. Take advantage of multiple CPU cores.

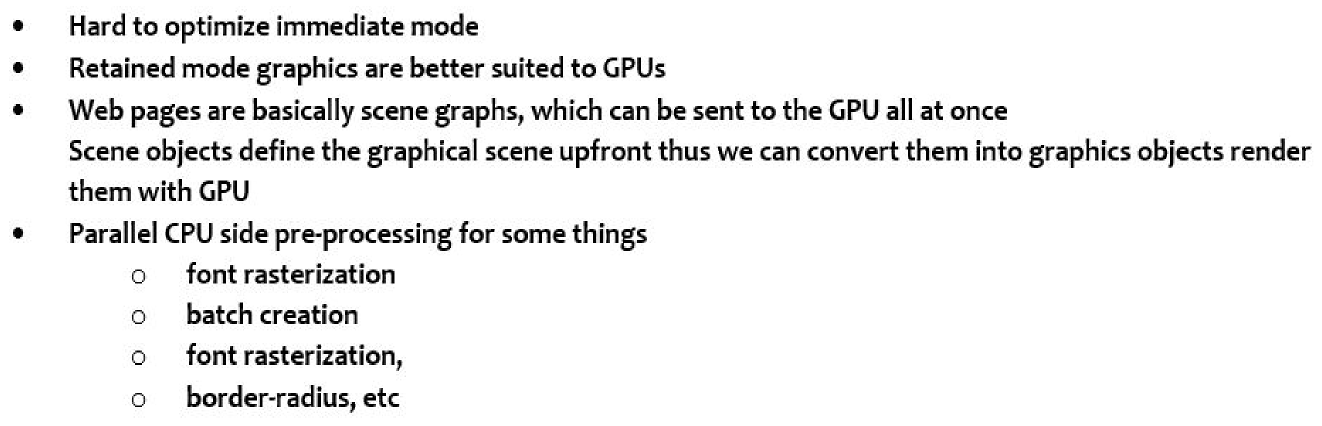
   - Many of the tasks in a game engine renderer are easily parallelized.

4. Specialized renderer for web content.

   - Not a general purpose vector graphics API.
   - Complex items such as canvas can be rendered on CPU.
   - Make the common case fast.

5. Draw only what's on screen

   - Aggressively cull items.
   - Redraw each frame (when there is something to be updated - not a constant fps like games!).
   - Cache results (such as vertex buffers, rasterized glyphs) where possible between frames.
   - Take advantage of knowledge of entire scene up front to optimize drawing.



- **Hard to optimize immediate mode**
- **Retained mode graphics are better suited to GPUs**
- **Web pages are basically scene graphs, which can be sent to the GPU all at once**
  Scene objects define the graphical scene upfront thus we can convert them into graphics objects render them with GPU
- **Parallel CPU side pre-processing for some things**
  - **font rasterization**
  - **batch creation**
  - **font rasterization,**
  - **border-radius, etc**

4.6.2 Architecture

The renderer runs through a number of steps each time a layout completes. The backend thread receives messages (such as add_image, new_layout). When it has built a new frame it notifies the compositor by sending a message with the relevant information (texture cache updates, pre-compiled vertex buffers, and render batches).

When scrolling an existing layout, steps 1 and 2 are skipped. Since items from steps 6 and 7 are cached between frames, these are typically very quick to execute during scrolling.

There is currently no caching of render items, display lists and batches when a new layout is received - this is a (potentially large) optimization for the future.

4.7 Constellation.

Each constellation instance can for now be thought of as a single tab or window, and manages a pipeline of tasks that accepts input, runs JavaScript against the DOM, performs layout, builds display lists, renders display lists to tiles and finally composites the final image to a surface.

4.7.1 Pipeline

The pipeline consists of four main tasks:

Script

Script's primary mission is to create and own the DOM and execute the JavaScript engine. It receives events from multiple sources, including navigation events, and routes them as necessary. When the content task needs to query information about layout it must send a request to the layout task.

Layout

Layout takes a snapshot of the DOM, calculates styles, and constructs the main layout data structure, the flow tree. The flow tree is used to calculate the layout of nodes and from there build a display list, which is sent to the render task.

Renderer

The renderer receives a display list and renders visible portions to one or more tiles, possibly in parallel.

Compositor

The compositor composites the tiles from the renderer and sends to the screen for display. As the UI thread, the compositor is also the first receiver of UI events, which are generally immediately sent to content for processing (although some events, such as scroll events, can be handled initially by the compositor for responsiveness).

Two complex data structures are involved in multi-task communication in this pipeline: the DOM and the display list. The DOM is communicated from content to layout and the display list from layout to the renderer. Figuring out an efficient and type-safe way to represent, share, and/or transmit these two structures is one of the major challenges for the project.

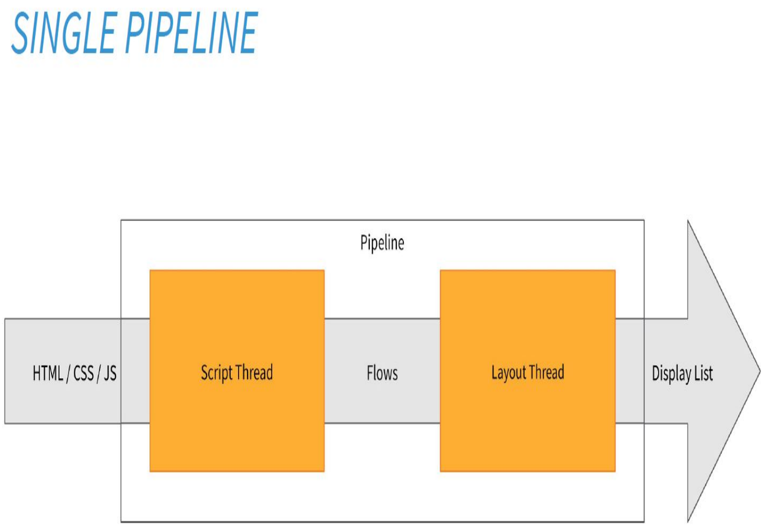


Figure 4.7: Single Pipeline

Above we isolate script and layout thread. The script is fed into script thread which produces a flow tree which is processed by the layout tree which results in a display list - a sequence of high-level drawing commands created by the layout task.

In case of cross orgin I-frames as shown,Each frame gets its own script-layout thread which gets processed in parallel entirely speeding up the process.

4.7.2 Threading Architecture

4.7.3 Benefits

- Iframes dont block one another
- Layout and Script wont block each other
- Failures in frames can be handled independently.

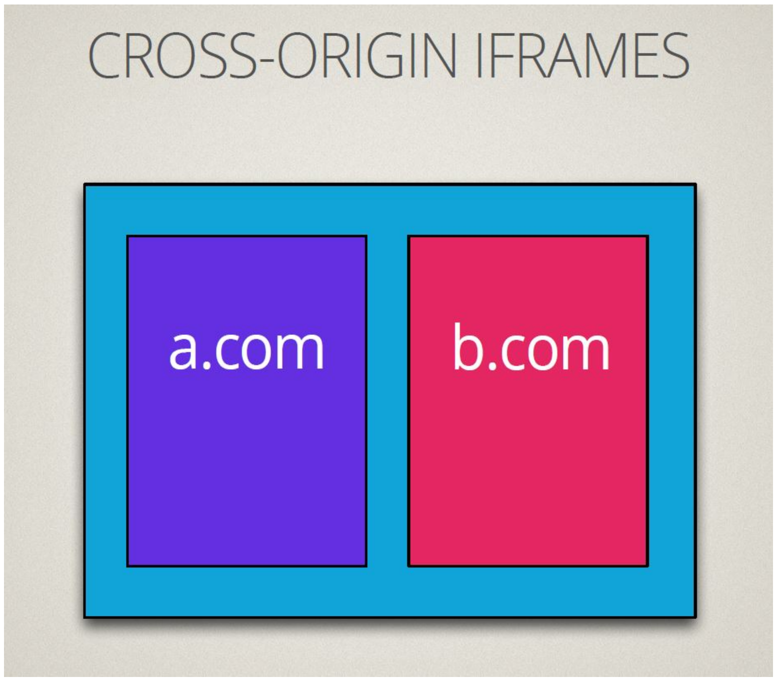


Figure 4.8: Cross Orgin I frames

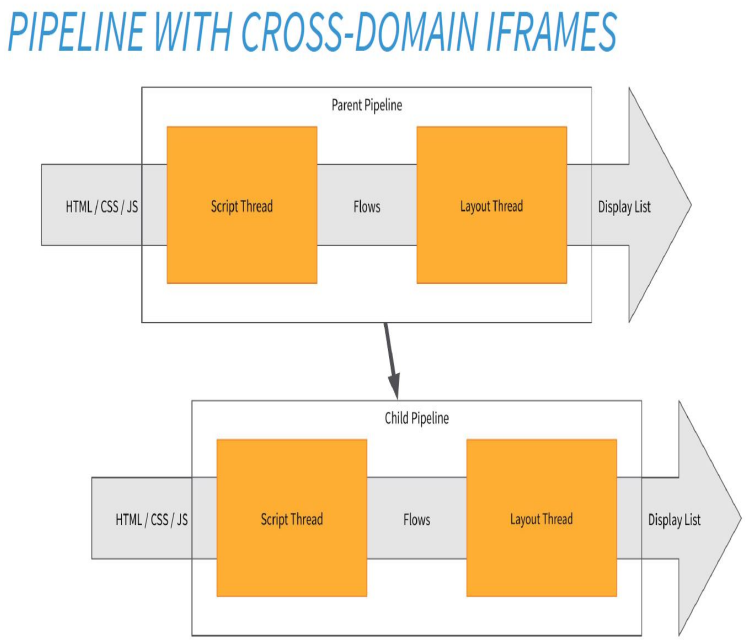


Figure 4.9: Cross Orgin I frame Pipeline

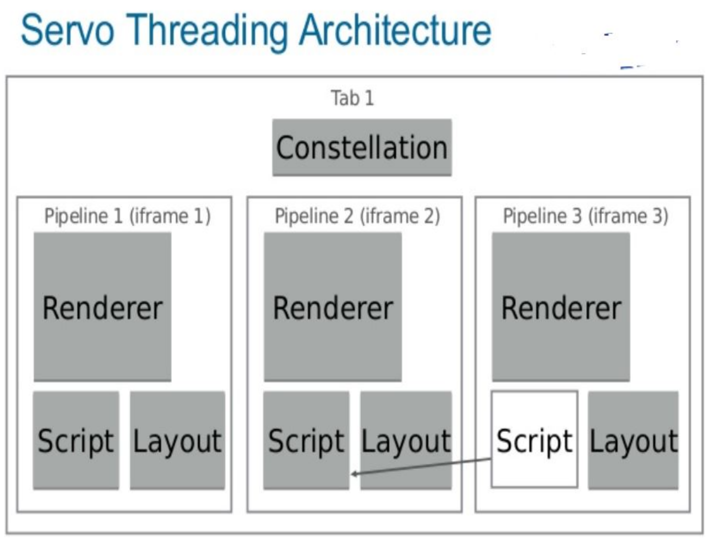


Figure 4.10: Threading Architecture

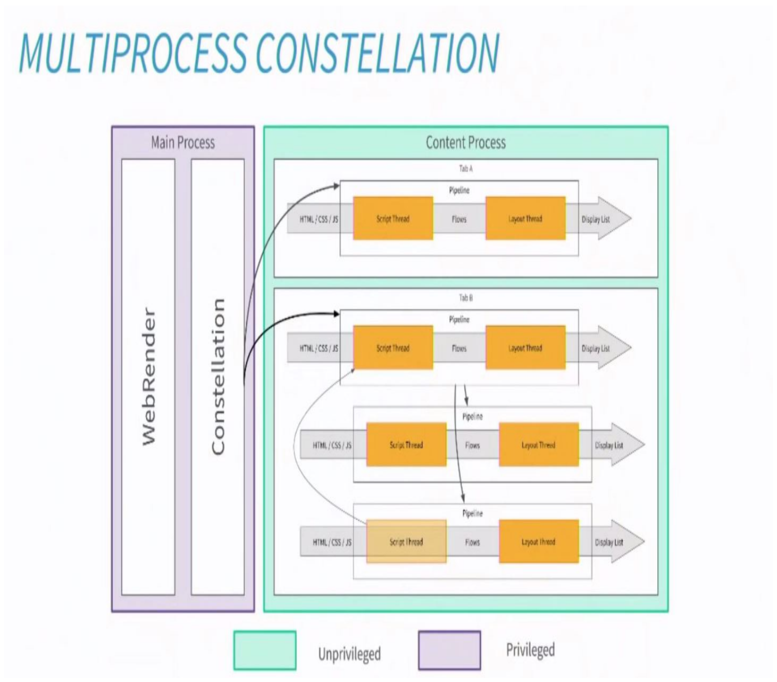


Figure 4.11: Multi-process Constellation

Chapter 5

Performance

5.1 Servo vs Gecko

5.1.1 Wikipedia

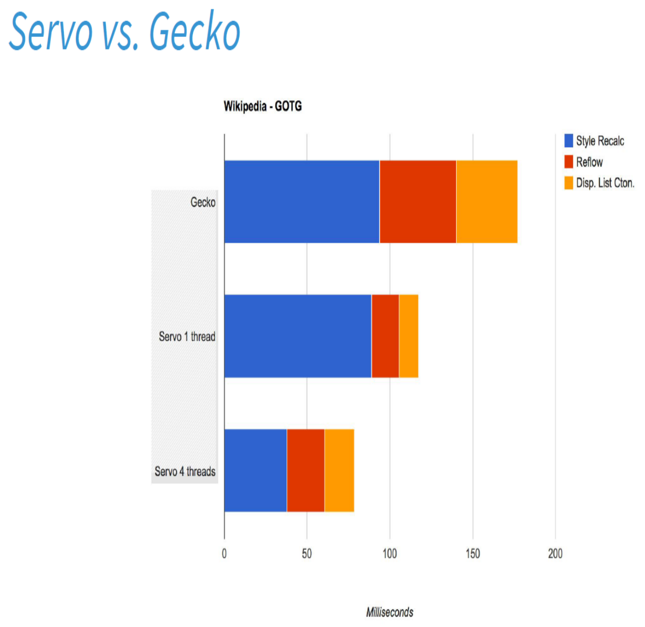


Figure 5.1: Servo vs Gecko - Wikipedia

5.1.2 Reddit

Running cores at a lower voltage reduces performance by 30% but power usage by 40%.

Servo can make up this performance loss through parallelism, achieving the same performance with 40% less power

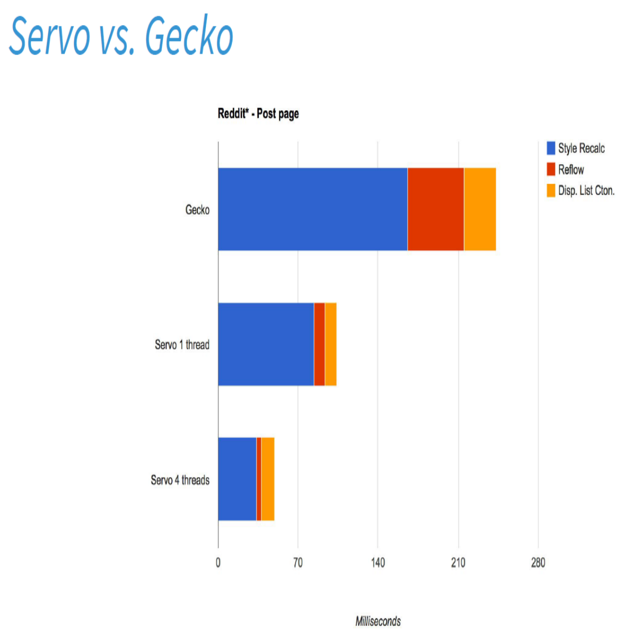


Figure 5.2: Servo vs Gecko - Reddit

Chapter 6

Conclusion

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, they are using a new systems programming language,Rust [RUS]. For parallelism and power, they scale across a wide variety of hardware by building either data- or task-parallelism, as appropriate, into each part of the web platform. Additionally, they 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 800k 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, Mozilla 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.

References

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

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

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

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

[5] 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, HotPar12, Berkeley, CA, 2012. USENIX Association.

[6] 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. 271280.

[7] 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.

[8] 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. 711720.

[9] Servo Today and Tomorrow - Jack Mofitt - Mozilla Research 2016

Appendices

Appendix A

Base Paper

Engineering the Servo Web Browser Engine using Rust 2016 IEEE/ACM 38th IEEE International Conference on Software Engineering Companion
Parallel Performance-Energy Predictive Modeling of Browsers: Case Study of Servo :2016 IEEE 23rd International Conference on High Performance Computing