Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Generic JavaScript-to-WebAssembly wrapper

Bachelor Thesis

2021
Eva Herencsárová

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Generic JavaScript-to-WebAssembly wrapper

Bachelor Thesis

Study Programme:	Computer Science
Field of Study:	Computer Science
Department:	Department of Computer Science
Supervisor:	RNDr. Richard Ostertág, PhD.
Consultant:	Dr. Dipl.-Ing. Andreas Haas, Bakk. techn.

Bratislava, 2021
Eva Herencsárová

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Eva Herencsárová
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Generic JavaScript-to-WebAssembly wrapper
Všeobecná „obalová funkcia“ pre WebAssembly funkcie volané z JavaScriptu

Anotácia: V8 je interpreter/kompilátor pre WebAssembly a JavaScript v prehliadači Google Chrome. V8 skompiluje pre každú WebAssembly funkciu volanú z JavaScriptu špecifickú „obalovú funkciu“. Tieto špecifické „obalové funkcie“ sú potrebné, aby sme vedeli prekladať medzi jednotlivými typmi v jazykoch JavaScript a WebAssembly. Kompilácia špecifických „obalov“ zaberie nejaký čas a prostriedky, ktoré by sme chceli ušetriť.

Cieľ: Cieľom tejto práce je navrhnúť a implementovať všeobecnú „obalovú funkciu“, ktorá by bola použiteľná pre ľubovoľnú WebAssembly funkciu volanú z JavaScriptu. Tým pádom by odpadla predkompilácia špecifických „obalových funkcií“.

Vedúci: RNDr. Richard Ostertág, PhD.
Konzultant: Dr. Dipl. Ing. Andreas Haas, Bakk. techn.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: prof. RNDr. Martin Škoviera, PhD.

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Dátum zadania: 05.10.2020

Dátum schválenia: 05.10.2020 doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Eva Herencsárová
Study programme:	Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Bachelor´s thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Generic JavaScript-to-WebAssembly wrapper

Annotation: V8 is the WebAssembly and JavaScript engine in Google Chrome browser. At the moment, V8 compiles a specific wrapper for WebAssembly functions that are called from JavaScript. These specific wrappers are needed because we need to translate back and forth between JavaScript and WebAssembly types. Compiling these wrapper functions takes some time and resources that we would like to save.

Aim: The goal of this thesis is to design and implement a generic wrapper for WebAssembly functions so that this wrapper can be used to call arbitrary WebAssembly functions from JavaScript; therefore, avoiding pre-compilation of specific wrappers.

Supervisor:	RNDr. Richard Ostertág, PhD.
Consultant:	Dr. Dipl. Ing. Andreas Haas, Bakk. techn.
Department:	FMFI.KI - Department of Computer Science
Head of department:	prof. RNDr. Martin Škoviera, PhD.
Assigned:	05.10.2020
Approved:	05.10.2020

doc. RNDr. Daniel Olejár, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Acknowledgments:

As the thesis was partially an internship project, special thanks goes to my host, Andreas Haas, co-host, Thibaud Michaud, and the WebAssembly Runtime Team for this wonderful opportunity and the rewarding experience. During the internship, I had the pleasure to work alongside superb software engineers who helped and supported me to successfully finish the project and acquire valuable knowledge.

Furthermore, I would like to thank my supervisor, RNDr. Richard Ostertág, PhD., for valuable advice, guidance and support.

Abstrakt

JavaScript interpreter/kompilátor je jedným zo základných podsystémov v architektúre webových prehliadačov, ktorý parsuje a spúšťa JavaScript kód, vďaka čomu sú webové stránky interaktívne. WebAssembly je relatívne nový jazyk, ktorý sa tiež dá spúšťať v moderných prehliadačoch. V prehliadači Google Chrome interpretrom/kompilátorom pre JavaScript a aj WebAssembly je V8. Keďže V8 vie spúšťať kód napísaný v oboch spomínaných jazykoch, je možné aj zavolať WebAssembly funkcie z JavaScriptu. Pôvodne V8 skompiloval špecifické wrapper funkcie pre rôzne WebAssembly funkcie volané z JavaScriptu, ktoré zaisťovali vhodnú konverziu medzi typmi týchto dvoch jazykov. V tejto práci sme úspešne navrhli a implementovali jednu všeobecnú wrapper funkciu pre WebAssembly funkcie volané z JavaScriptu vo V8. Vďaka našej novej všeobecnej wrapper funkcii, ktorú môžeme použiť pre štandardné WebAssembly funkcie volané z JavaScriptu, odpadla predkompilácia špecifických wrapper funkcií, čím sme úspešne ušetrili čas a prostriedky.

Kľúčové slová: V8, Google Chrome, WebAssembly, JavaScript, wrapper funkcie, volacie konvencie

Abstract

The JavaScript engine is an essential subsystem in the web browser's architecture that parses and executes JavaScript code embedded in web pages making them interactive. WebAssembly is a relatively new language that can be also run in modern web browsers. In Google Chrome, V8 is the JavaScript and WebAssembly engine. As V8 can execute code written in both of these languages, it is also possible to have function calls from JavaScript to WebAssembly. Before, V8 compiled specific wrapper functions for WebAssembly functions called from JavaScript to ensure conversion between the two languages' types. In this bachelor's thesis, we successfully designed and implemented a generic JavaScript–to–WebAssembly wrapper function in V8. The new, generic wrapper can be used for standard type WebAssembly functions when called from JavaScript. By using the generic wrapper, compilation of specific wrapper functions was avoided, therefore, we improved the compilation time and saved resources.

Keywords: V8, Google Chrome, WebAssembly, JavaScript, wrapper functions, calling conventions

Preface

This bachelor's thesis was partially created during a summer internship at Google Switzerland GmbH.

Google Switzerland GmbH has all copyright, design rights, other proprietary rights, rights of ownership and use to all Computer Programs conceived or developed during the term of the internship, i.e. the code related to the generic JavaScript-to-WebAssembly wrapper function. This internship project was initiated by the WebAssembly Runtime Team to further improve Google Chrome browser's JavaScript and WebAssembly engine, V8.

Contents

Introduction

One of the most widely used software applications are web browsers. Web browsers retrieve information from web servers, display it and allow client-side computation. Throughout these processes, we expect high performance from web browsers.

JavaScript is a well-known scripting language that enables web developers to add interactivity to web applications. WebAssembly is a relatively new, low-level language that can be run alongside JavaScript in modern web browsers. A key goal of WebAssembly is to serve as a compilation target for other programming languages (e.g. to take an application written in C++ like Google Earth, compile it to WebAssembly, and run it in the browser). [2, 5]

In this bachelor's thesis, we will focus on V8, Google's open source high-performance JavaScript and WebAssembly engine used in the most popular modern web browser, Google Chrome [21].
We will investigate how JavaScript and WebAssembly interact with each other, more precisely, the way we call WebAssembly functions from JavaScript in V8. Due to value representation differences in these two languages, we have to ensure proper conversions between the value types. For these conversions, specialized wrapper functions are compiled based on the number and types of function parameters and return values.

As a result of this thesis, we want to design and implement one generic wrapper function that can be used to call arbitrary WebAssembly function from JavaScript. Therefore, pre-compilation of the specific wrapper functions can be avoided by compiling only the generic wrapper.
Note that the generic wrapper is more complex compared to the specific ones resulting in run-time performance overhead. Thus, for optimal efficiency results, tier-up strategies should be considered to ensure that for frequently called functions the specific wrappers are compiled.

Chapter 1

Background

In this chapter, we will talk about the architecture of web browsers to understand and locate the JavaScript and WebAssembly engine subsystem. Furthermore, we will study the main concepts of WebAssembly and emphasize the advantages of using it in the JavaScript environment in web browsers.

1.1 Web browsers

Modern web browsers are constantly evolving and gaining complexity, therefore, efficiency is one of the most important factors in their software quality.

If we look back, the first web browser, created by Tim Berners-Lee in 1991, was text-only and it was able to display only simple static web pages written using HyperText Markup Language (HTML). This simple web browser parsed the HTML tags to present the encoded information and also served as an HTML editor.
To attract more users, graphical browsers were created and numerous innovations were introduced setting a new standard for web browsers. [12]

1.1.1 The browser architecture

Nowadays, modern browsers are highly sophisticated, displaying web pages from HTML, CSS, JavaScript and handle interaction with ease.

At a high-level, we can divide the web browser into several components (subsystems) as in Figure 1.1. Each subsystem has an important role in the information retrieval and displaying process. We can briefly describe these roles: [12]

1. *User Interface* – Basically, it is everything inside the browser window excluding the area where the requested page appears. It includes the address bar for the URL address, toolbar, buttons, etc.

2. *Browser Engine* – Manipulates the actions between the *User Interface* and the *Rendering Engine*. It is responsible for loading the given URL address and supporting simple browsing actions (forward, back or reload).

3. *Rendering Engine* – Responsible for displaying HTML, CSS and XML documents with embedded content such as images.

4. *Networking* – Handles network calls and file transfer.

5. ***JavaScript Engine*** – The JavaScript Engine that parses and executes the JavaScript code embedded in web pages.

6. *XML Parser* – Used to parse XML documents. It is often a generic and reusable browser component with a standard, well-defined interface compared to the HTML parser which is commonly tightly integrated with the *Rendering Engine*.

7. *Display Backend* – Used for drawing and windowing primitives (UI widgets, set of fonts), may be tied with the OS.

8. *Data Persistence* – Stores and manages user data related to the browsing session (cookies, bookmarks, etc.).

As mentioned above, the complexity of modern web pages is increasing and new features are being added. WebAssembly is one of the new "extensions", that is why we need a WebAssembly compiler subsystem too. In Google Chrome, V8 is the JavaScript and WebAssembly engine.
We will learn more about WebAssembly in Section 1.2, Google Chrome in Subsection 1.1.2 and about V8 in Chapter 2.

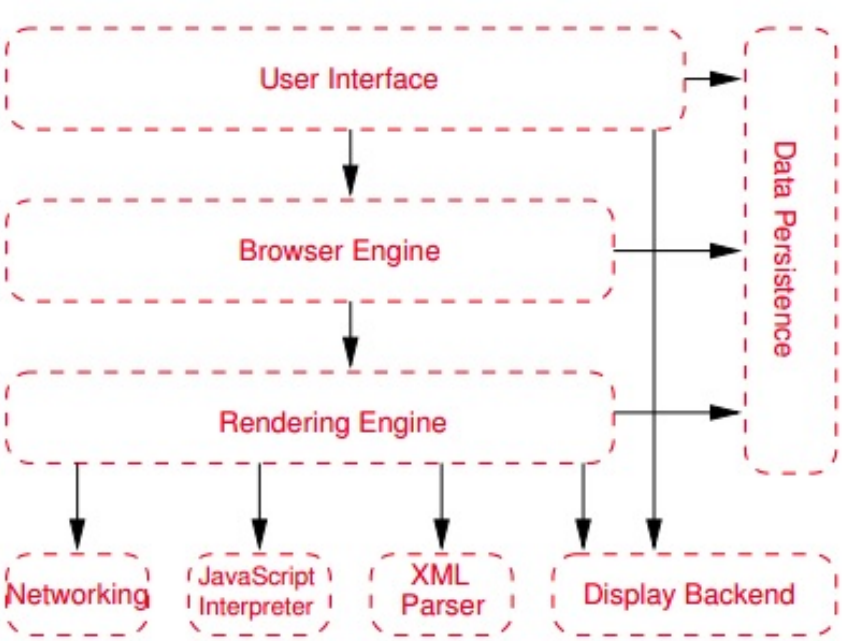

Figure 1.1: *Reference architecture for web browsers* [12]. *We can see the major subsystems and the dependencies among them.*

1.1.2 Google Chrome

Google Chrome is a web browser based on an open-source software project called Chromium. In Figure 1.2 [7], we can see Google Chrome's subsystems mapped to the reference architecture for web browsers from Figure 1.1.

This popular web browser was first released in 2008. Unlike most web browsers back then, Google Chrome used multi-process architecture. Multi-process architecture means separate rendering engine processes for browser tabs. This was a crucial step for better stability and performance, because the rendering engine's work became complex over time. If the rendering engine crashed in one web application, the multi-architecture allowed the browser to continue with the remaining responsive web applications. [3]

V8, the new JavaScript engine, was also released in 2008 which compiled JavaScript code directly into machine code. Experimental support for WebAssembly was available in V8 and Chromium in 2016. [4]

Throughout the years, Google Chrome has gained popularity becoming today the most popular web browser in the world accounting for about 70% of the global desktop internet browser market share [15].

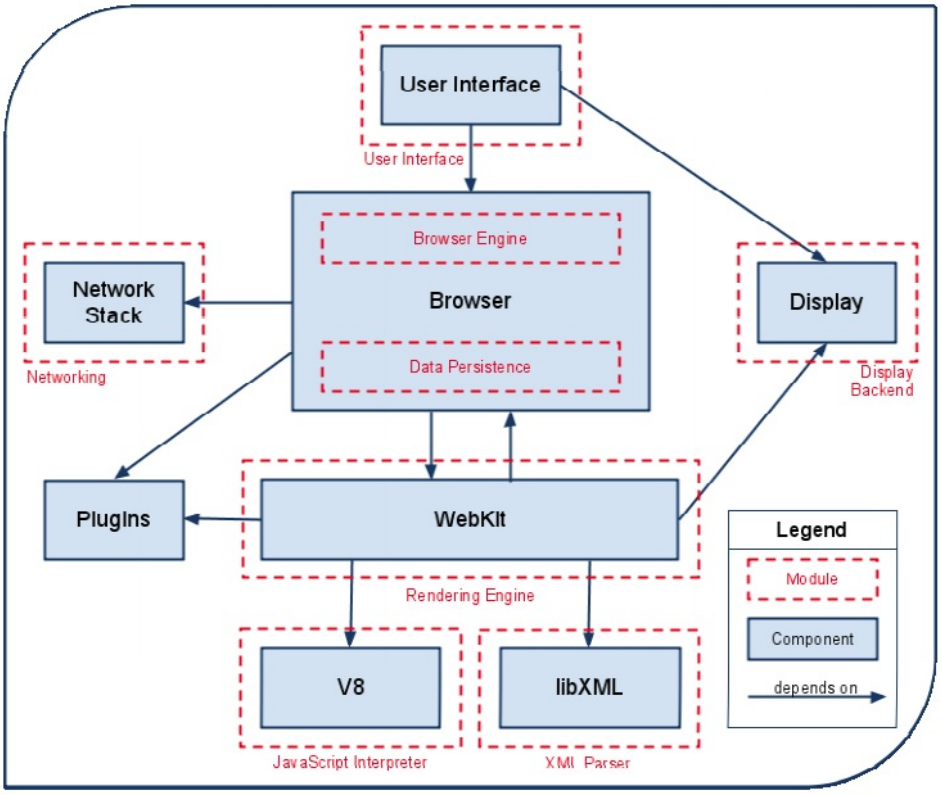

Figure 1.2: *Conceptual architecture of Google Chrome* [7]

1.2 WebAssembly

In this section, we will learn about WebAssembly to understand relevant concepts for the thesis and the importance of the work around this new technology.

1.2.1 Introduction to WebAssembly

WebAssembly is a binary instruction format for a stack-based virtual machine. It was designed as a portable compilation target for programming languages such as C, C++, Rust, etc., enabling near native speed on the web. WebAssembly is supported in all four major browsers: Google Chrome, Mozilla Firefox, Edge and Safari. [2]

In Figure 1.3, we can see an illustration about the compilation process of a high-level language (e.g. C, C++, Rust) into machine code of the given architecture (e.g. x86, ARM). WebAssembly, i.e. *wasm*, stands there as the next step after creating the intermediate representation (*IR*). However, WebAssembly is not the same as other kinds of assembly, as it is a language for a virtual machine, not an existing physical machine, and thanks to its portability capabilities, it has a much more direct mapping to actual machine code compared to JavaScript. Therefore, the web browser can easily turn the virtual instructions of WebAssembly into the real processor's code format. [9]

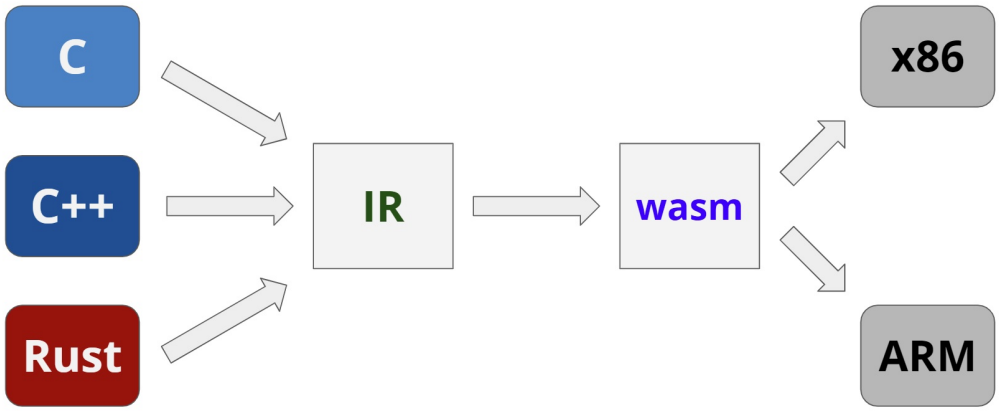

Figure 1.3: *Illustration of the compilation of high-level languages (C, C++, Rust) into machine code with WebAssembly (wasm)* [9]

1.2.2 Design goals

WebAssembly was created with the following goals to be fulfilled: [5, 2, 18]

- **Efficient, fast and portable**
  WebAssembly takes advantage of common hardware capabilities on a wide range of platforms aiming to achieve near native code performance. Furthermore, it can be embedded in browsers or integrated in other software environments.

- **Safe and secure**
  Similarly to JavaScript, data corruption and security breaches are prevented by creating a memory-safe, sandboxed execution environment. When running in web browsers, WebAssembly enforces the browser's same-origin and permissions security policies.

- **Readable and debuggable**
  Despite of being a low-level assembly-like language, WebAssembly has a textual representation allowing debugging, viewing and writing programs by hand. We can easily convert textual format into binary and also the other way around. However, WebAssembly is primarily intended to be an effective compilation target and not written by hand.

- **Part of the web platform**
  As one of WebAssembly's purposes is to run on the web platform, it is designed to be compatible with the web technologies and maintain backwards compatibility.

1.2.3 WebAssembly as part of the web platform

JavaScript is a flexible, dynamically typed high-level language, furthermore, it is the most popular scripting language used to create dynamic web pages. Undoubtedly, JavaScript is a powerful tool, however, when using it for performance demanding tasks (such as 3D games or image/video editing), performance issues may occur.
To avoid such issues, we should consider using WebAssembly for these more intensive use cases. Indeed, WebAssembly is great to deal with bottlenecks, i.e. where near-native performance is necessary.
However, we should not use WebAssembly as a replacement for JavaScript. These two languages are intended to be used together on the web platform, so we can benefit from both languages' strong points. [18]

To understand the performance differences between WebAssembly and JavaScript, we first have to explain these two languages' compilation/interpretation phase. Let's look at Figure 1.4. (Note: we will study V8 more precisely in the next chapter.)

In V8, *Ignition* is the JavaScript interpreter pipeline, *Liftoff* is the WebAssembly compiler and *TurboFan* is the optimizing compiler. When interpreting JavaScript with *Ignition*, analytic data of the code execution is being collected. This data is used for *TurboFan* to optimize the code by compiling certain parts that are used more often (e.g. when a function is called many times in a loop). However, as we mentioned before, JavaScript is dynamically typed, so the optimized code may not be applicable in every case (e.g. some properties of an object were dynamically changed). In this case, we have to deoptimize. It can be clearly seen that in some cases continuous optimization and deoptimization may lead to performance issues. [20]

Below, we can see a simple function using the += operation in a for loop. Let's assume that the *arr* array consists of 100 integers. After a number of iterations, the code "warms up" and the baseline compiler will compile this part of the code. However, the *arr* array may also contain a string, so the optimized compiled code cannot be used. Therefore we have to deoptimize. [8]

```
function arraySum(arr) {
  var sum = 0;
  for (var i = 0; i < arr.length; i++) {
    sum += arr[i];
  }
}
```

On the other hand, WebAssembly is statically typed, therefore, when compiling WebAssembly, the deoptimization can never happen, because *TurboFan* can only begin after *Liftoff* has finished. This means that WebAssembly has a predictable performance in contrast to JavaScript. [20]

It is important to mention that WebAssembly, aside from being predictable, is also

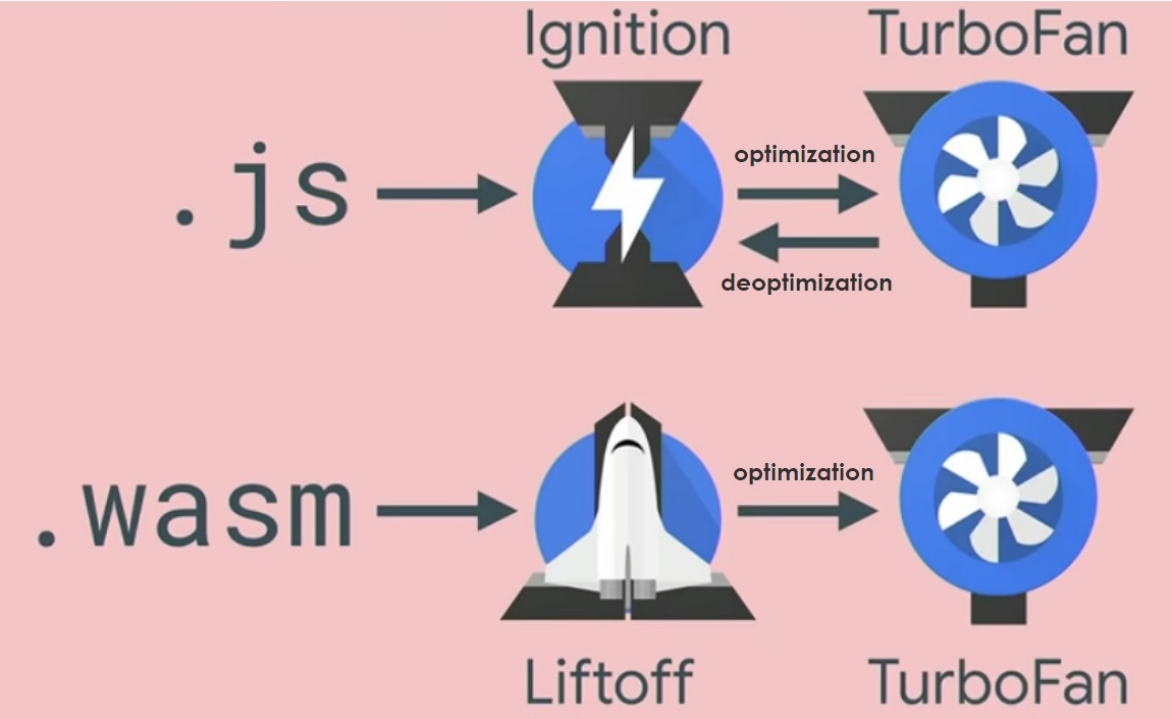

Figure 1.4: *Interpreting JavaScript (with V8's interpreter, Ignition and the optimizing compiler, TurboFan) and compiling WebAssembly (with V8's WebAssembly compiler, Liftoff and TurboFan)* [20]

faster. Its binaries are lighter than the textual JavaScript files, therefore, WebAssembly gets loaded, parsed and executed in a shorter amount of time compared to JavaScript. [19]

The predictability problem can be seen on Figure 1.5 where a benchmark written both in WebAssembly and JavaScript was run in different browsers. Overall, besides WebAssembly being faster thanks to its design, we can see that the scattering of the measurements of the JavaScript code is significantly larger than for the WebAssembly code. [20]

1.2.4 Main concepts

In this subsection, we introduce the key concepts of WebAssembly to understand how it runs in the browser.

In WebAssembly, the unit of deployment, compilation and loading is the *module*. The *module* is stateless and it represents a WebAssembly binary. This WebAssembly *module* consists of several components, it defines types, functions, tables, memories, global and local variables, imports and exports, and initializing values.
An instantiated *module* is paired with all the state it uses at runtime (including a *memory, table, etc.*).

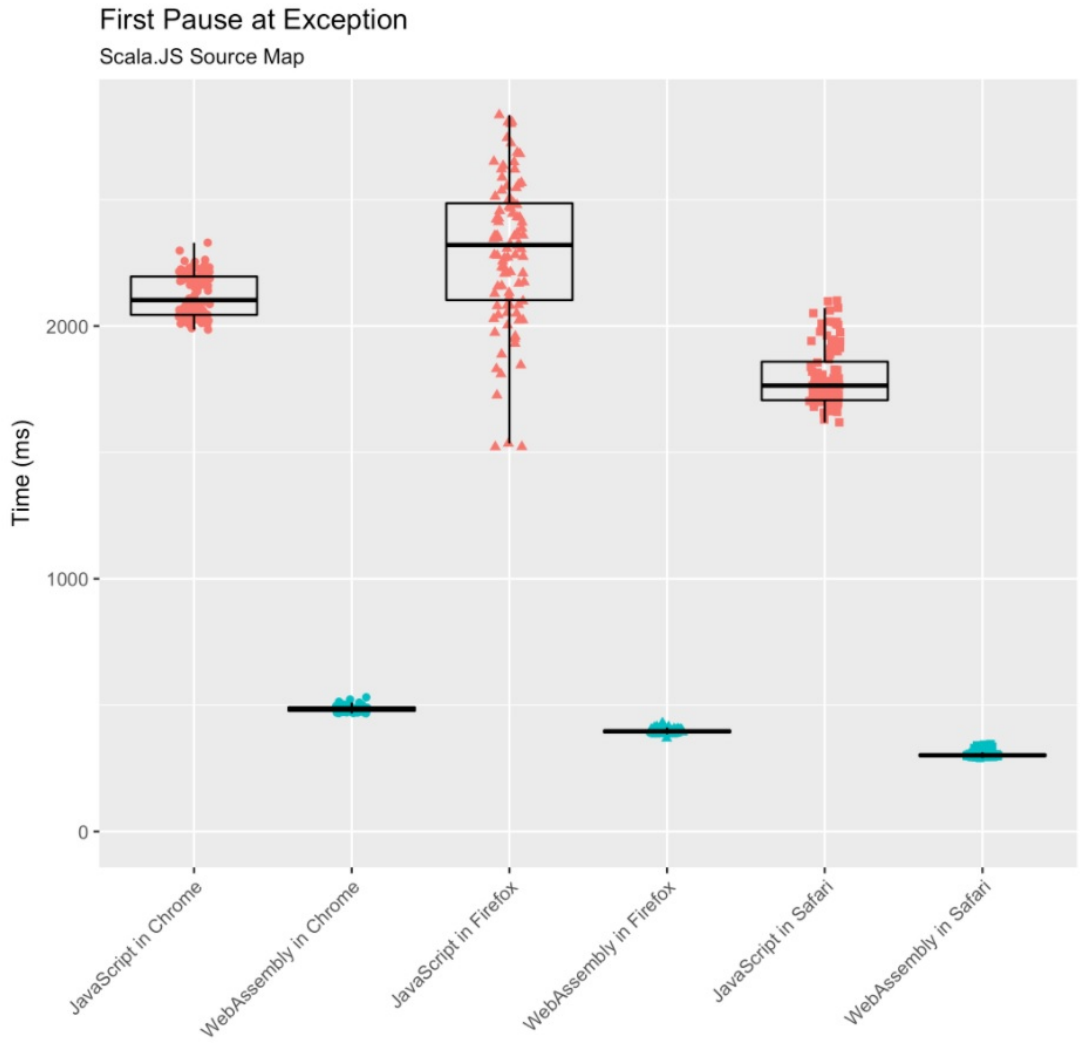

Figure 1.5: *Benchmark written in JavaScript and also WebAssembly for speed comparison in various browsers by Nick Fitzgerlad* [11]

Let's look more precisely at the components that are relevant for our generic wrapper [5, 18]:

- ***Types***
  For our thesis, the most relevant types are the 4 standard *number types*: *i32, i64, f32, f64* (32-bit and 64-bit integer and floating-point value, respectively).
  This component also defines another relevant type, the *function type* that specifies the *signature* of functions. The *signature* determines the number and type of parameters and return values of the given function.
  (To learn more about other WebAssembly types visit [5].)

- ***Memory***
  Defines a vector of linear memory (raw, uninterpreted bytes) that is being accessed by WebAssembly's low-level memory access instructions.

- ***Table***
  Defines a vector of references (e.g. to functions), so it is possible to map to values outside of the WebAssembly module (e.g. JavaScript objects, operating system file-handles). These are references that could not be stored in the *memory* (for safety and portability reasons).

- ***Exports and imports***
  This component defines a set of *exports* that are accessible from the host environment after instantiation, and a set of *imports* that are required for instantiation. Exportable and importable definitions are *functions*, *tables*, *memories* and *globals* (i.e. global variables of the module).
  In our case, the host environment is JavaScript. To sum up, when given a WebAssembly instance, JavaScript code can synchronously call its *exported functions*. These *exported functions* are exposed as normal JavaScript functions. Similarly, JavaScript functions – passed as *imports* to a WebAssembly module – can also be synchronously called by WebAssembly code.
  Note that our generic wrapper will be used for *exported functions* – WebAssembly functions called from JavaScript. A generic wrapper for JavaScript functions called from WebAssembly could also be created.

Chapter 2

V8

As mentioned before, JavaScript is used to make web pages interactive. To ensure smooth user experience, we expect JavaScript to run fast in the browser. JavaScript engines (i.e. interpreters or compilers) are responsible for guaranteeing high performance.
In this chapter, we will talk about V8, Google's open source JavaScript and WebAssembly engine used also in Google Chrome. We will explain the key concepts and topics related to the implementation of our generic wrapper.

2.1 Main concepts

To process JavaScript source code, we need a tool to parse and translate JavaScript into machine code that can be executed.

A *compiler* takes the whole source code, runs through it, makes smaller optimizations if possible, and translates it into the target language – in our case into machine code. After the translation phase, the machine code can be executed with the given inputs. However, compiling long source code can take some time, therefore, it may not be convenient for web applications. [6]

An *interpreter*, on the other hand, directly executes the operations specified in the source program on the given inputs. That is, instead of translating the whole source code and only after that executing it, an *interpreter* translates a single statement (e.g. one line of source code) into machine code, executes it and moves further. This means that with an *interpreter* we start executing the source code faster, however, the machine code created by an *interpreter* is not optimized and therefore slower. [6]

To benefit from both of these traditional language processors, we can use *JIT* – *just-in-time* – compilers. A *JIT* compiler compiles the source code into machine code during program execution, i.e. at run time, and may use a *profiler* (or *monitor*). The *profiler* collects additional analytic data during runtime, and remembers how often a

given part of code (e.g. functions, methods) is being executed and what types were used. Based on these runtime observations, we can categorize the functions as not so frequent, as *warm* or as *hot*. To increase speed, *warm* functions will be compiled by the baseline compiler, and whenever the same code with the same types is about to be executed, the stored compiled version will be used. Similarly, whenever functions are being called very often – are *hot* – these functions will be sent to the optimizing compiler that compiles them into even faster, optimized version of machine code that will be stored and reused for the same types.
Note that different browsers have slightly different way of the above mentioned *JIT* compilation, but they follow the basic idea we have just explained. [8]

V8, as all modern JavaScript engines, is a *JIT* compiler written in C++. To process and execute JavaScript V8 uses:

- ***Ignition*** – interpreter pipeline. First, the JavaScript source code is being compiled to concise *bytecode* by the baseline compiler. *Bytecode* is a low-level program code compiled from source code (e.g. JavaScript) and it is designed for an *interpreter*. Furthermore, *bytecode* takes up significantly less space, it is around 50% to 25% the size of equivalent baseline machine code. In the next step, the bytecode is being executed by a high-performance interpreter or it is processed by *TurboFan* resulting in an architecture specific machine code. [16, 17]

- ***TurboFan*** – the optimizing compiler for both WebAssembly and JavaScript. It is built around a concept called *Sea of Nodes* [10]. This means that *TurboFan's* intermediate representation is a graph where nodes (that represent operations) are linked together with different types of edges representing dependencies in data-flow and control-flow.

In Figure 2.1, we can see the the interpreter pipeline *Ignition* and the optimizing compiler, *Turbofan*. Furthermore, we can observe the compilation process of WebAssembly. In V8, ***Liftoff*** is the baseline compiler that compiles WebAssembly source code into unomptimized machine code very fast. In the next steps, ***TurboFan*** generates highly optimized machine code that runs closer to near-native speeds. [13, 14, 21]

2.2 Pointer compression

The generic wrapper will have to deal with compressed and decompressed values. Let's explain the idea behind this technique.

In 2014, Chrome switched from being a 32-bit process to a 64-bit process achieving better security and stability. However, this improvement comes at a cost of memory

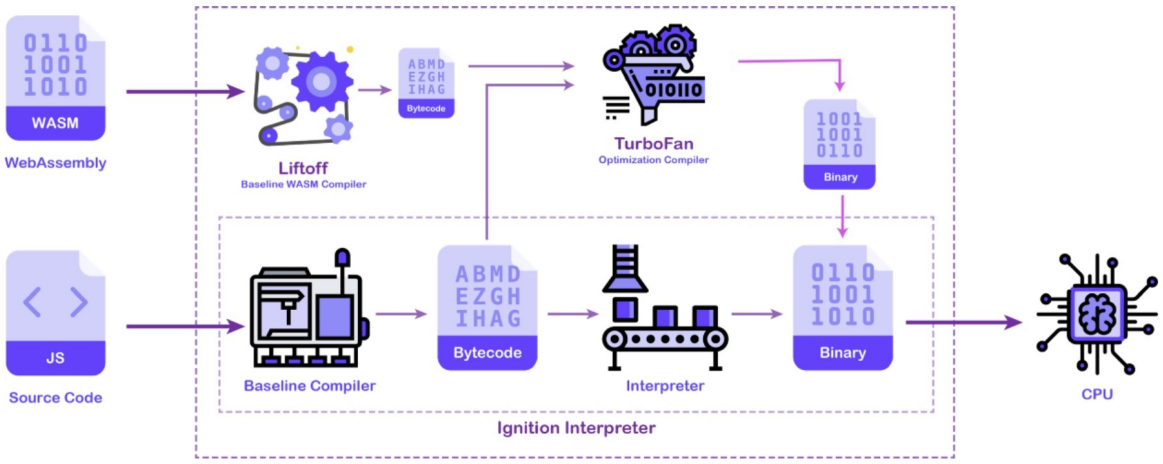

Figure 2.1: *V8's JavaScript and WebAssembly compilation pipeline* [14]

consumption. Each pointer occupies eight bytes instead of four. With pointer compression in V8, a memory usage of a 32-bit application with performance of a 64-bit one was achieved. The main idea of pointer compression is to store 32-bit offsets from some *base* address instead of storing a whole 64-bit value. [21]

In V8, JavaScript values are represented as objects and allocated on the V8 heap, allowing us to represent them as pointers to the objects. Pointer tagging technique is used to store additional information in V8 pointers. This allows us to create two types of values based on a tag bit: real pointers and values that represent integers. [21]

As V8 allocates objects in the heap only at word-aligned addresses, the 2 (or 3, depending on the machine word size) least significant bits can be used for tagging.
V8 uses the least significant bit to distinguish small integers, called *Smis*, from heap object pointers:

- if the bit equals to 1 then it is a pointer,
- if the bit equals to 0 then it is a *Smi*.

Furthermore, for pointers the second least significant bit is used to distinguish strong references from weak ones.
Note: this means that on 32-bit architectures a *Smi* can only carry a 31-bit payload and a pointer can use 30 bits for addresses.
Having 4 GB addressable space (with 4 byte granularity due to world-alignment) may not sound too good, but V8 in Google Chrome already has a 2 GB or 4 GB limit on the size of the V8 heap (even on 64-bit architectures). [21]

In our wrapper function, we will be working with compressed and decompressed values. For this, we will always have to use the proper instructions, and bear in mind which parts of the register we are using and are rewritten after the instruction (e.g. lower 32-bits or full 64-bits).

2.3 Compiling specific wrappers

As already mentioned, to transition from JavaScript to WebAssembly, we need conversion functions – *wrapper* functions (*wrappers* for short) – that ensure proper conversions between the value types. That is, after instantiating the WebAssembly module and calling an exported WebAssembly function from JavaScript, we have to use a specific wrapper based on the function signature.

Before the generic wrapper, the function that was responsible for creating the JavaScript to WebAssembly wrappers was ***BuildJSToWasmWrapper***. It is a method in the *WasmWrapperGraphBuilder* class which is used to build wrapper functions related to WebAssembly. Note that we have to compile a wrapper once per signature (and not once per exported function). We will use this function as a reference for the logic to transition from JavaScript to WebAssembly in our generic wrapper. *TurboFan*, the optimizing compiler, is used to generate the wrapper functions. As the wrappers are created by the optimizing compiler, the compilation time takes longer. Hence, being able to compile only one generic wrapper function could significantly reduce the startup time.

Let's briefly explain the idea behind the *BuildJSToWasmWrapper* method[1].
The *sig_* variable stands for the function signature that contains the number and types of parameters and return values. Note that support for WebAssembly functions with multi-returns was added recently. However, we will not support these types of functions in our wrapper (we plan to implement it in the future).
As *TurboFan*'s intermediate representation is a graph, the *BuildJSToWasmWrapper* method will create linked nodes. First, we start by creating the *starting node* of the control-flow and other relevant nodes for *TurboFan* (*js_closure, js_context*):

```
1  void BuildJSToWasmWrapper(bool is_import) {
2      const int wasm_count = static_cast<int>(sig_->parameter_count());
3      const int rets_count = static_cast<int>(sig_->return_count());
4
5      // Build the start and the JS parameter nodes.
6      SetEffectControl(Start(wasm_count + 5));
7
8      // Create the js_closure and js_context parameters.
9      Node* js_closure =
10         graph()->NewNode(mcgraph()->common()->Parameter(
11                              Linkage::kJSCallClosureParamIndex, "%closure"),
12                          graph()->start());
13     Node* js_context = graph()->NewNode(
14         mcgraph()->common()->Parameter(
15             Linkage::GetJSCallContextParamIndex(wasm_count + 1), "%context"),
16         graph()->start());
```

[1] *https://source.chromium.org/chromium/_/chromium/v8/v8.git/+/462378addab907ab7452d38baccb27fe2ef7289a:src/compiler/wasm-compiler.cc;l=6002;drc=4f50c554bae694d56b34238f95b52f4315649dd3;bpv=1;bpt=0*

On line 22, we load into the *instance_ node* a pointer to the WebAssembly instance. It will be important for our generic wrapper for accessing definitions (like the memory) of the WebAssembly module.

```
17    // Create the instance_node node to pass as parameter. It is loaded from
18    // an actual reference to an instance or a placeholder reference,
19    // called {WasmExportedFunction} via the {WasmExportedFunctionData}
20    // structure.
21    Node* function_data = BuildLoadFunctionDataFromExportedFunction(js_closure);
22    instance_node_.set(
23        BuildLoadInstanceFromExportedFunctionData(function_data));
```

After checking if the signature is valid on line 24, we convert all the JavaScript parameters into compatible types in WebAssembly:

```
24    if (!wasm::IsJSCompatibleSignature(sig_, enabled_features_)) {
25      // Throw a TypeError. Use the js_context of the calling javascript
26      // function (passed as a parameter), such that the generated code is
27      // js_context independent.
28      BuildCallToRuntimeWithContext(Runtime::kWasmThrowTypeError, js_context,
29                                    nullptr, 0);
30      TerminateThrow(effect(), control());
31      return;
32    }
33
34    const int args_count = wasm_count + 1;  // +1 for wasm_code.
35    base::SmallVector<Node*, 16> args(args_count);
36    base::SmallVector<Node*, 1> rets(rets_count);
37
38    // Convert JS parameters to wasm numbers.
39    for (int i = 0; i < wasm_count; ++i) {
40      Node* param = Param(i + 1);
41      Node* wasm_param = FromJS(param, js_context, sig_->GetParam(i));
42      args[i + 1] = wasm_param;
43    }
```

Next, we prepare for calling the WebAssembly function by setting a flag (*ThreadInWasm flag*) on line 45 (later unsetting on line 70). We may call an exported function that was imported and it is handled separately (on lines 47-53), but we will ignore it and not use our generic wrapper for this special case (we plan to implement it in the future).

Otherwise, on lines 54-67, we create nodes for calling the WebAssembly function:

```
44    // Set the ThreadInWasm flag before we do the actual call.
45    BuildModifyThreadInWasmFlag(true);
46
47    if (is_import) {
48      // Call to an imported function.
49      // Load function index from {WasmExportedFunctionData}.
50      Node* function_index =
51          BuildLoadFunctionIndexFromExportedFunctionData(function_data);
52      BuildImportCall(sig_, VectorOf(args), VectorOf(rets),
53                      wasm::kNoCodePosition, function_index, kCallContinues);
54    } else {
55      // Call to a wasm function defined in this module.
56      // The call target is the jump table slot for that function.
57      Node* jump_table_start =
58          LOAD_INSTANCE_FIELD(JumpTableStart, MachineType::Pointer());
59      Node* jump_table_offset =
60          BuildLoadJumpTableOffsetFromExportedFunctionData(function_data);
```

```
61        Node* jump_table_slot = graph()->NewNode(
62            mcgraph()->machine()->IntAdd(), jump_table_start, jump_table_offset);
63        args[0] = jump_table_slot;
64
65        BuildWasmCall(sig_, VectorOf(args), VectorOf(rets), wasm::kNoCodePosition,
66                      nullptr, kNoRetpoline);
67      }
68
69      // Clear the ThreadInWasm flag.
70      BuildModifyThreadInWasmFlag(false);
```

Finally, we handle the conversion of the return values back to JavaScript values:

```
71      Node* jsval;
72      if (sig_->return_count() == 0) {
73        jsval = BuildLoadUndefinedValueFromInstance();
74      } else if (sig_->return_count() == 1) {
75        jsval = ToJS(rets[0], sig_->GetReturn());
76      } else {
77        int32_t return_count = static_cast<int32_t>(sig_->return_count());
78        Node* size =
79            graph()->NewNode(mcgraph()->common()->NumberConstant(return_count));
80
81        jsval = BuildCallAllocateJSArray(size, js_context);
82
83        Node* fixed_array = BuildLoadArrayBackingStorage(jsval);
84
85        for (int i = 0; i < return_count; ++i) {
86          Node* value = ToJS(rets[i], sig_->GetReturn(i));
87          STORE_FIXED_ARRAY_SLOT_ANY(fixed_array, i, value);
88        }
89      }
90      Return(jsval);
91      if (ContainsInt64(sig_)) LowerInt64(kCalledFromJS);
92  }
```

Chapter 3

Design and implementation of the generic wrapper

Compiling specific wrappers takes some time and resources at startup that we would like to save by creating a generic wrapper. In this chapter, we will describe the design and implementation workflow of the generic wrapper and the challenges we faced.

In Section 2.3, we described the logic to create specific wrappers with the ***BuildJSToWasmWrapper*** method. Another relevant method is the **Generate_CEntry**[1] that implements a JavaScript to C wrapper function as a built-in function written in V8 macro assembly. Both methods show us different ways to generate machine code in V8. As we needed the flexibility offered by the macro assembler, the generic wrapper was written in V8 macro assembly as a built-in function (similarly to the *Generate_CEntry*) for architecture *x64*. (Note that we plan to implement the generic wrapper for other architectures too in the future.) We began with a wrapper that supports only the simplest signature, and continued by gradually adding support for other signatures.

3.1 Main structure of the generic wrapper

Our first step was to introduce a wrapper that could handle functions with the simplest signature – no parameters and no return values.

First, we created a V8 flag that enables our generic wrapper, and set up the compilation to use the generic wrapper only for WebAssembly functions with no parameters and no return values if our new flag is set to true. To test our wrapper, we wrote a test in JavaScript that creates a WebAssembly module, imports a simple JavaScript function (*import_func*) that only changes the value of a global variable, and after in-

[1] *https://source.chromium.org/chromium/_/chromium/v8/v8.git/+/1a1aa77644c24808d3356293ac0602cd23506b57:src/builtins/x64/builtins-x64.cc;l=2708;drc=26824a285cc1536e19d5738bff1c4cbfa6bc69ab;bpv=1;bpt=0*

stantiating the WebAssembly module, it calls the *import_func* and checks if the value stored in the global variable is correct. The test can be seen in Listing 3.1.

```
(function testGenericWrapper() {
  print(arguments.callee.name);
  let builder = new WasmModuleBuilder();
  let sig_index = builder.addType(kSig_v_v);
  let func_index = builder.addImport("mod", "func", sig_index);
  builder.addFunction("main", sig_index)
    .addBody([
      kExprCallFunction, func_index
    ])
    .exportFunc();

  let x = 12;
  function import_func() {
    x = 20;
  }

  builder.instantiate({ mod: { func: import_func } }).exports.main();
  assertEquals(x, 20);
})();
```

Listing 3.1: Our first function to test the generic wrapper functionality for WebAssembly functions with no parameters and no return values[1].

To make our new test pass, we started implementing the generic wrapper in V8 macro assembly for the *x64* architecture based on the ***BuildJSToWasmWrapper*** method from Section 2.3. In our generic wrapper, the main task was to set up the *stack* for the WebAssembly function which is a part of the run-time memory.

Generally, when a procedure is called, a code sequence known as the *calling sequence* is responsible for transferring the control to the procedure. To ensure proper transfer and state restoration of the machine after the call, we have to save records on the *stack*. We create a *stack frame* (the *frame pointer* or *base pointer* points to the start of the stack frame) and save all the necessary values like the return address, frame pointer, parameters, local data, etc. [6]

3.1.1 Register types

A register is the fastest memory type to hold a value. Usually, we only have a limited number of registers, so all other values should reside in memory. Instructions where the registers are the operands are always faster and shorter than those with operands in memory. Therefore, for high performance, it is very important to utilize the registers efficiently. [6]

When assigning values to registers or doing instructions with register operands, we have to pay attention to several things, e.g.:

- We always have to use registers that support the instruction we want to use.

[1] *https://chromium-review.googlesource.com/c/v8/v8/+/2307240/17/test/mjsunit/wasm/generic-wrapper.js*

- There are special purpose registers that are reserved and should not be used.
- Use the calling conventions (how arguments are passed from a calling procedure to the callee, and return values passed back to the caller).
- Be aware of what part of the register we are using (e.g. lower 16-bits, lower 32-bits, full 64-bits, ...)

Based on the V8 source code [1], for platform *x64* the following facts and statements apply[1]:

- The **general purpose registers** are: *rax, rbx, rcx, rdx, rsp, rbp, rsi, rdi, r8, r9, r10, r11, r12, r13, r14, r15*, where *rsp, rbp, r10, r13* are not allocatable.
- The **non-allocatable general purpose registers** should contain:
  - *rsp*: contains the stack pointer that points to the value that is on the top of the stack,
  - *rbp*: contains the base (or frame) pointer that points to the start of the stack frame,
  - *r10*: scratch register that is used by the macro assembler,
  - *r13*: root register that stores the *isolate_root* which is a pointer to the *Isolate*. The *Isolate* is the V8 instance containing the global memory. (We can think about it as a V8 instance containing the global state for a browser tab.)
- The allocatable double registers are: *xmm0, ..., xmm14*. They should be used for floating-point values and instructions with floating-point operands.
- For the calling conventions:
  - *rax, rdx, rcx, rbx, r9*: contain the first 5 integer (*i32* or *i64*) parameters, respectively,
  - *xmm1, xmm2, xmm3, xmm4, xmm5, xmm6*: contain the first 6 floating-point *(f32* or *f64)* parameters, respectively,
  - *rax* and *xmm0*: contain the first integer and floating-point return value, respectively.

[1] *https://source.chromium.org/chromium/_/chromium/v8/v8.git/+/554b7ee535b3eca3f041dc45d00de5cbd490729c:src/codegen/x64/register-x64.h;bpv=1;bpt=0;drc=ab5470212eb90c9aa8b3d2541bf51f8f2b4bc992* and *https://source.chromium.org/chromium/_/chromium/v8/v8.git/+/554b7ee535b3eca3f041dc45d00de5cbd490729c:src/wasm/wasm-linkage.h;l=31;drc=e7cb911a93f372a6dbbeacab8f0a615c0c58a9c4;bpv=1;bpt=0*

3.1.2 The structure and the stack layout

The first version of our wrapper did not deal with parameters nor return values. Our task was to correctly set up the stack, call the proper function and restore the stack afterwards. Let's walk through the code of our wrapper[1] and explain each step.

When the generic wrapper is called, the return address is automatically saved on the stack. The first thing we had to do was to save the address of the start of the frame in the proper register, *rbp*. However, *rbp* contains the start of the frame of the caller procedure and we did not want to lose it. Therefore, we pushed the value in *rbp* onto the stack before setting it to the proper value. We also pushed a specific constant value onto the stack (*JS_TO_WASM*), the *frame marker*, that indicates the type of the frame. Saving and setting the value in *rbp*, and saving the *frame marker* on the stack was done by the *EnterFrame* macro instruction on line 3:

```
1  void Builtins::Generate_GenericJSToWasmWrapper(MacroAssembler* masm) {
2    // Set up the stackframe.
3    __ EnterFrame(StackFrame::JS_TO_WASM);
```

The next step was to load into registers a pointer to the *WasmExportedFunctionData* object (*function_data*) that contains metadata to the exported WebAssembly function, and a pointer to the WebAssembly instance (*wasm_instance*).
We expected that register *rdi* had contained the *closure*, an address to a V8 heap object. Based on the address of the *closure* and constant offsets, we loaded the *function_data* on lines 6-16, and the (*wasm_instance*) on lines 19-23. (See below.)
Note: when assigning a register to a *Register* type (like assigning *rdi* to *Register closure* on line 4), we create an alias for the register for better readability. Similarly, *no_reg* deletes this alias improving readability.

```
4    Register closure = rdi;
5    Register shared_function_info = rbx;
6    __ LoadAnyTaggedField(
7        shared_function_info,
8        MemOperand(
9            closure,
10           wasm::ObjectAccess::SharedFunctionInfoOffsetInTaggedJSFunction()));
11
12   Register function_data = shared_function_info;
13   __ LoadAnyTaggedField(
14       function_data,
15       MemOperand(shared_function_info,
16                  SharedFunctionInfo::kFunctionDataOffset - kHeapObjectTag));
17   shared_function_info = no_reg;
18
19   Register wasm_instance = rsi;
20   __ LoadAnyTaggedField(
21       wasm_instance,
22       MemOperand(function_data,
23                  WasmExportedFunctionData::kInstanceOffset - kHeapObjectTag));
```

[1] *https://chromium-review.googlesource.com/c/v8/v8/+/2307240/17/src/builtins/x64/builtins-x64.cc*

Based on the *wasm_instance* and constant offsets, we had to load the address to the *thread_in_wasm_flag* like in the *BuildJSToWasmWrapper* method before calling the proper WebAssembly function – see on lines 24-34. We set the flag to true on line 34 by writing value 1 on the address that is stored in register *thread_in_wasm_flag*.

```
24  int isolate_root_offset =
25      wasm::ObjectAccess::ToTagged(WasmInstanceObject::kIsolateRootOffset);
26
27  // Set thread_in_wasm_flag.
28  Register isolate_root = rdx;
29  __ movq(isolate_root, MemOperand(wasm_instance, isolate_root_offset));
30  Register thread_in_wasm_flag_addr = isolate_root;
31  __ movq(
32      thread_in_wasm_flag_addr,
33      MemOperand(isolate_root, Isolate::thread_in_wasm_flag_address_offset()));
34  __ movl(MemOperand(thread_in_wasm_flag_addr, 0), Immediate(1));
35  isolate_root = no_reg;
```

The next step was to call the proper function. We got its address by several loads and adding constant offsets to the *wasm_instance* or the *function_data*.
We had to do a conversion on lines 52-53 from *Smi* (small integer) to a 64-bit integer (due to pointer compression).

```
36  Register jump_table_start = thread_in_wasm_flag_addr;
37  __ movq(jump_table_start,
38          MemOperand(wasm_instance,
39                     wasm::ObjectAccess::ToTagged(
40                         WasmInstanceObject::kJumpTableStartOffset)));
41  thread_in_wasm_flag_addr = no_reg;
42
43  Register jump_table_offset = function_data;
44  __ DecompressTaggedSigned(
45      jump_table_offset,
46      MemOperand(
47          function_data,
48          WasmExportedFunctionData::kJumpTableOffsetOffset - kHeapObjectTag));
49  function_data = no_reg;
50
51  // Change from smi to int64.
52  __ sarl(jump_table_offset, Immediate(1));
53  __ movsxlq(jump_table_offset, jump_table_offset);
54
55  Register function_entry = jump_table_offset;
56  __ addq(function_entry, jump_table_start);
57  jump_table_offset = no_reg;
58  jump_table_start = no_reg;
```

On line 59, just before calling the WebAssembly function on line 62, we pushed the pointer to the WebAssembly instance onto the stack, because we needed this value after the WebAssembly function had finished. We restored it right after on line 66 so we could load the pointer to the *Isolate* (*isolate_root*) again to unset the *thread_in_wasm_flag* on line 75.

```
59  // Save wasm_instance on the stack.
60  __ pushq(wasm_instance);
61
62  __ call(function_entry);
63  function_entry = no_reg;
```

```
64
65    // Restore wasm_instance.
66    __ popq(wasm_instance);
67
68    // Unset thread_in_wasm_flag.
69    isolate_root = rdx;
70    __ movq(isolate_root, MemOperand(wasm_instance, isolate_root_offset));
71    thread_in_wasm_flag_addr = r8;
72    __ movq(
73        thread_in_wasm_flag_addr,
74        MemOperand(isolate_root, Isolate::thread_in_wasm_flag_address_offset()));
75    __ movl(MemOperand(thread_in_wasm_flag_addr, 0), Immediate(0));
```

Even though we did not have to deal with any return values in this version of our wrapper, we had to set the register that should contain the first integer return value, *rax*, to a constant that indicates undefined value:

```
76    Register return_reg = rax;
77    __ movq(return_reg,
78            MemOperand(isolate_root, IsolateData::root_slot_offset(
79                                         RootIndex::kUndefinedValue)));
```

Lastly, on line 81, we restored *rbp* to point to the start of the caller frame and removed our wrapper's frame marker with the macro instruction *LeaveFrame*, and transferred program control to the return address (with a parameter that indicates how many slots should be removed after the return address) on line 83:

```
80    // Deconstrunct the stack frame.
81    __ LeaveFrame(StackFrame::JS_TO_WASM);
82
83    __ ret(kSystemPointerSize);
84  }
```

In figure 3.1, we can see the stack layout during this first version of our generic wrapper. The *caller frame slots* contain a pointer to the receiver (JavaScript object) and the function arguments (as JavaScript values) that must be removed before transferring the program control to the caller function. In the first version of the wrapper, the *caller frame slots* section contained just one slot, a pointer to the receiver. (This is the reason we had to remove one slot that has a size of *kSystemPointerSize* on line 83.)

The next slot (in the direction of growth) is occupied by the implicitly saved return address followed by the base pointer and the frame marker in the next two slots.

During the wrapper function, we may save several values on the stack (spill slots) that had to be removed before the wrapper had finished. In the first version of our wrapper (for the no parameter and no return value WebAssembly functions), we used only one spill slot to save the address to the WebAssembly instance before calling the WebAssembly function.

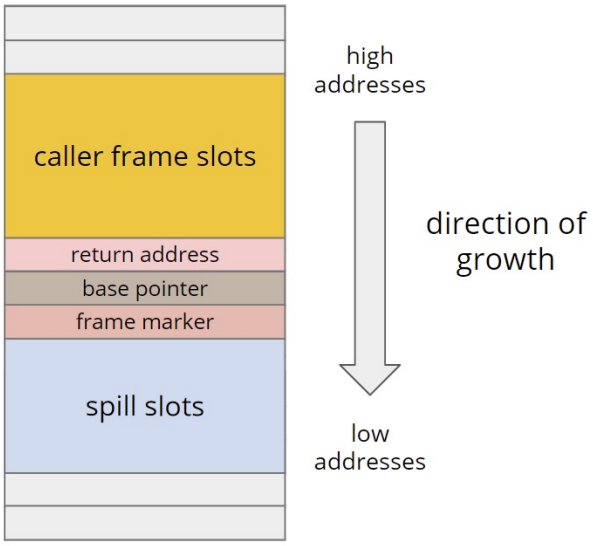

Figure 3.1: *General stack layout for the generic wrapper*

There were several smaller challenges we had to overcome when implementing the first version of our generic wrapper.

1. **Dealing with pointer compression.** As we mentioned in Section 2.2, we had to consider how the given value is represented and which instructions to use to access and manipulate with the values properly.

2. **Saving values on the stack.** We realized that after we call to another function (line 62), the function may change the values stored in the registers that we used before. We had to reconsider which values are needed after the call and use the stack to save them.

3. ***Compile time* vs *runtime*.** We realized that some values change during runtime (after our generic wrapper was compiled). At first, our wrapper could not react properly to a flag that enabled trap handling and could change at runtime, therefore, we had to change some conditions to handle runtime changes properly[1].

3.2 Handling one 32-bit integer parameter

The next natural step to extend our wrapper's functionality was adding support for parameter handling. In this section, we will describe how we made the generic wrapper work also for WebAssembly functions with one *i32* parameter and no return value[2].

[1] *https://chromium-review.googlesource.com/c/v8/v8/+/2307240/17/src/runtime/runtime-wasm.cc*

[2] *https://chromium-review.googlesource.com/c/v8/v8/+/2339622*

We have already showed the general stack layout in Figure 3.1 and explained that the *caller frame slots* include the JavaScript receiver and JavaScript arguments. It can be clearly seen that we can easily access the parameters relatively to the base pointer stored in *rbp*. In Figure 3.2, we can see both of the possible orders of the JavaScript arguments.

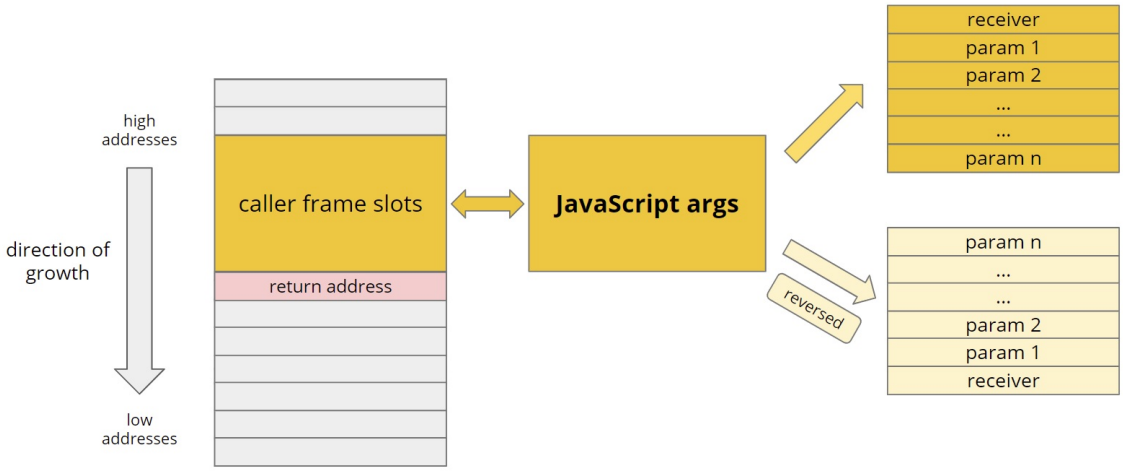

Figure 3.2: *The caller frame slots consist of the JavaScript receiver and the JavaScript arguments. The order of the arguments may be reversed depending on the V8_REVERSE_JSARGS macro. The advantage of the reversed order is that even if the number of provided parameters does not match the expected number of parameters, a function can easily access them.*

So, to handle the one parameter, we just moved the value from the proper slot to the register that should contain the first parameter, *rax*. If the arguments are not reversed, the proper slot is above the return address, in the direction of higher addresses (we handled both the not reversed and reversed arguments in the later versions of the wrapper):

```
// Param handling.
Register param = rax;
const int firstParamOffset = 2*kSystemPointerSize;
__ movq(param, MemOperand(rbp, firstParamOffset));
```

However, the value of the argument may be a *Smi* or a *non-Smi* (e.g. *valueOf()* JavaScript method that returns a 32-bit integer) depending on the tag bit. If it is a *Smi*, we used an instruction to convert it to a *i32*. Otherwise, if it is a *non-Smi*, we had to call a built-in function that handles the conversion. As the more frequent case is converting a *Smi*, we put the conversion of the *non-Smi* with the built-in function into the deferred code part. (Note: deferred code is the part of the code that is not

part of the main flow. We can jump to the deferred part from the main flow, e.g. to convert a value, and then jump back):

```
Label not_smi;
__ JumpIfNotSmi(param, &not_smi);

// Change from smi to int32.
__ SmiUntag(param);


...

// -------------------------------------------------------------------------
//                          Deferred code.
// -------------------------------------------------------------------------

// Handle the conversion to int32 when the param is not a smi.
__ bind(&not_smi);

__ pushq(wasm_instance);
__ pushq(function_data);
__ pushq(signature_type);
__ LoadAnyTaggedField(
    rsi,
    MemOperand(wasm_instance, wasm::ObjectAccess::ToTagged(
                                  WasmInstanceObject::kNativeContextOffset)));
// We had to prepare the parameters for the Call:
// put the value into rax, and the context to rsi.
__ Call(BUILTIN_CODE(masm->isolate(), WasmTaggedNonSmiToInt32),
        RelocInfo::CODE_TARGET);

__ popq(signature_type);
__ popq(function_data);
__ popq(wasm_instance);

__ jmp(&params_done);
```

In the listing above, we can see that before calling the built-in function for the *non-Smi* conversion, we had to save some values on the stack and then restore them (*pushq* and *popq* instructions) to prevent losing them after the built-in function.

So far we showed how to prepare the JavaScript argument for the WebAssembly function. However, there are two big challenges hiding we haven't discussed yet:

1. interpreting the signature,
2. adding garbage collection support.

3.2.1 Interpreting and accessing the signature

Inside our generic wrapper, we had to decide if we are dealing with a WebAssembly function with zero or with one *i32* parameter.
In the listing above, we saw that besides the *wasm_instance* and *function_data* we saved and restored a register with *signature_type* alias. For now the *signature_type* register stored only 0 or 1 depending on the number of parameters. To access the

number of parameters, we added a field to the *WasmExportedFunctionData* class which is metadata to the exported WebAssembly function[1]. See below on lines 5-7:

```
1 Handle<WasmExportedFunction> WasmExportedFunction::New(
2     Isolate* isolate, Handle<WasmInstanceObject> instance, int func_index,
3     int arity, Handle<Code> export_wrapper) {
4   ...
5   const wasm::FunctionSig* sig = instance->module()->functions[func_index].sig;
6   sig->parameters().empty() ? function_data->set_signature_type(0)
7                             : function_data->set_signature_type(1);
8   ...
9 }
```

We had already accessed the *WasmExportedFunctionData* object before in our wrapper to load the address of the WebAssembly instance. Therefore, similarly based on a constant offset we could load the address of the field that stores the information about the signature (lines 2-5). Next, based on its value, we could skip the parameter handling (lines 7-11) or load and convert the parameter[2]:

```
1   // Int signature_type gives the number of int32 params (can be only 0 or 1).
2   Register signature_type = r9;
3   __ SmiUntagField(
4       signature_type,
5       MemOperand(function_data, WasmExportedFunctionData::kSignatureTypeOffset - ↩
            kHeapObjectTag));
6
7   __ cmpl(signature_type, Immediate(0));
8
9   // In 0 param case jump through parameter handling.
10  Label params_done;
11  __ j(equal, &params_done);
```

Obviously, the generic wrapper must be able to decide any types of signatures. This was just a temporary solution for this version of the wrapper. We will discuss the design options later for the generic version.

3.2.2 Adding garbage collection support

Objects on the JavaScript heap that are no longer used are marked and later deleted. To avoid segmentation, the garbage collector (GC for short) manages the memory by compacting it. This means that the GC may move objects in the memory, therefore, all the references to the moved objects get updated.
Calling the WebAssembly function or built-in functions may trigger the GC and rearrange some objects (e.g. the address of the WebAssembly instance). Therefore, saving pointers to the objects by storing addresses on the stack before calling a function is not enough. We will have to iterate through pointers to the JavaScript heap objects

[1] *https://chromium-review.googlesource.com/c/v8/v8/+/2339622/6/src/wasm/wasm-objects.cc*

[2] *https://chromium-review.googlesource.com/c/v8/v8/+/2339622/6/src/builtins/x64/builtins-x64.cc*

stored on the stack and update them.

In the previously mentioned version of the wrapper, we had to save the value indicating the signature type and the pointer to the WebAssembly instance before calling the WebAssembly function. When we call the built-in for conversion, besides the above mentioned two values, we had to save a pointer to the *WasmExportedFunctionData* object (*function_data*). From these values only the value that indicates the signature types is not a heap object, so it must not be scanned. On the other hand, the slots storing the pointers to the WebAssembly instance and *WasmExportedFunctionData* must be scanned.

By placing these values on the top of the stack (in the direction of growth) and iterating through them was the easiest way to scan and update these slots. In Figure 3.3, we can see the stack layout which is convenient for this GC scanning.

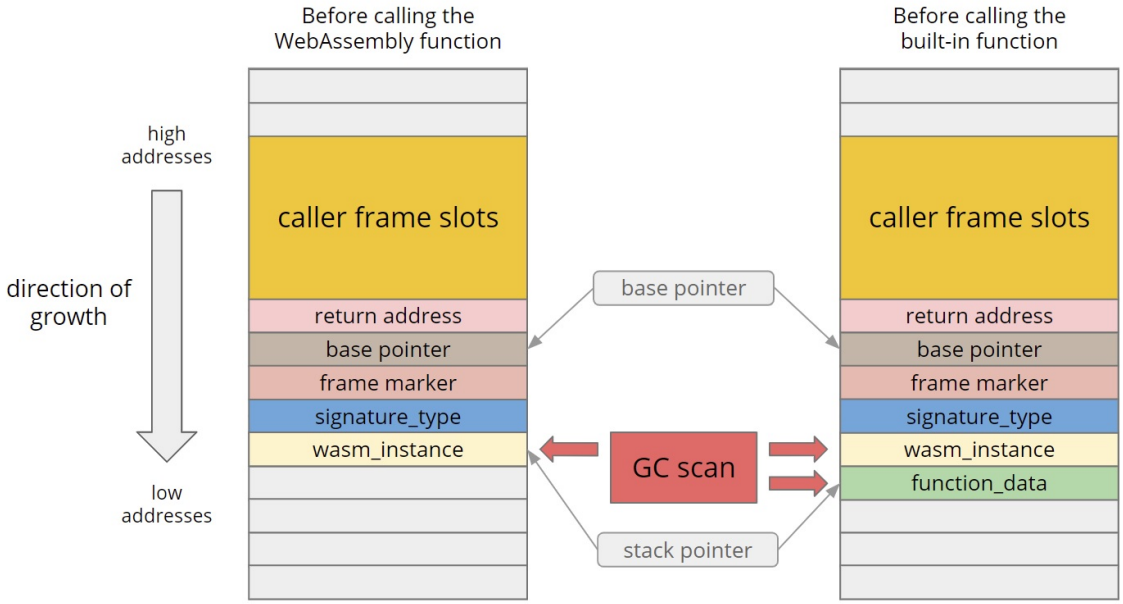

Figure 3.3: *Stack layout of the generic wrapper when the GC can be triggered*

We know that the base pointer points to the start of the frame (on the *base pointer* slot) of our wrapper function, and the stack pointer points to the top slot on the stack. Therefore, based on the base and stack pointer we could easily select the values we had to scan. We called the *VisitRootPointers* function (line 27) to scan the right half-open interval from the top stack slot (*spill_slot_base*) until the frame (base) pointer minus the size of 2 slots (*spill_slot_limit*)[1]:

[1] `https://chromium-review.googlesource.com/c/v8/v8/+/2351673/7/src/execution/frames.cc`

```
void JsToWasmFrame::Iterate(RootVisitor* v) const {
  Code code = GetContainingCode(isolate(), pc());
  //  GenericJSToWasmWrapper stack layout
  //  ------+-----------------+----------------------
  //        |  return addr    |
  //    rbp |- - - - - - - - -| <-fp() -------------|
  //        |  base pointer   |                     |
  //  rbp-p |- - - - - - - - -|                     |
  //        |  frame marker   |                     | no GC scan
  // rbp-2p | - - - - - - - - | <- spill_slot_limit |
  //        |  signature_type |                     |
  // rbp-3p |- - - - - - - - -|  -------------------|
  //        |      ....       |                     |
  //        |   spill slots   |                     | GC scan
  //        |      ....       |<- spill_slot_base   |
  //        |- - - - - - - - -|  -------------------|
  if (code.is_null() || !code.is_builtin() ||
      code.builtin_index() != Builtins::kGenericJSToWasmWrapper) {
    // If it's not the  GenericJSToWasmWrapper, then it's the TurboFan compiled
    // specific wrapper. So we have to call IterateCompiledFrame.
    IterateCompiledFrame(v);
    return;
  }
  FullObjectSlot spill_slot_base(&Memory<Address>(sp()));
  FullObjectSlot spill_slot_limit(
      &Memory<Address>(fp() - 2 * kSystemPointerSize));
  v->VisitRootPointers(Root::kTop, nullptr, spill_slot_base, spill_slot_limit);
}
```

3.3 Design options for the signature interpretation in the generic wrapper

In our generic wrapper, we have to access the signature to properly handle the WebAssembly function call. In this section, we will describe the signature interpretation for our wrapper and the other options we considered.

The function signature is represented by a C++ class. Its fields are: the number of return values, the number of parameters, and a pointer to an array that contains the types of both the return values and parameters. [1]

As we already mentioned before, we can easily access the *WasmExportedFunctionData* object in our wrapper. In the older version of our wrapper, we added an integer field to this class that indicated if the function had zero or one integer parameter, and we accessed it through adding a constant offset to the address of the *WasmExportedFunctionData*. To support any type of signature in our wrapper, we decided to save a pointer to the original signature as a field in the *WasmExportedFunctionData*. We had to use a *Foreign pointer* (wrapper class around a raw pointer) instead of a raw pointer field, because the GC interprets raw pointers as JavaScript heap objects[1]:

[1] *https://chromium-review.googlesource.com/c/v8/v8/+/2369178/9/src/wasm/wasm-objects.cc*

```
Handle<WasmExportedFunction> WasmExportedFunction::New(
    Isolate* isolate, Handle<WasmInstanceObject> instance, int func_index,
    int arity, Handle<Code> export_wrapper) {
  ...
  const wasm::FunctionSig* sig = instance->module()->functions[func_index].sig;
  Handle<Foreign> sig_foreign =
      isolate->factory()->NewForeign(reinterpret_cast<Address>(sig));
  ...
  function_data->set_signature(*sig_foreign);
  ...
}
```

We also considered creating our own array containing all the information from the signature class, but then we would have to think about its memory management.

Other alternatives we considered to access the signature:

- **Accessing the signature similarly to the *sig()*[1] method:**
  We could have accessed the signature in our wrapper similarly to the following method based on the WebAssembly instance:

  ```
  const wasm::FunctionSig* WasmExportedFunction::sig() {
    return instance().module()->functions[function_index()].sig;
  }
  ```

  The disadvantage of this approach is that we need many indirections to load the signature based on the WebAssembly instance. Furthermore, *functions* is a C++ *vector* with dynamically changing size, so accessing the signature would have been more complicated.

- **Change the current implementation of the signature globally:**
  Unfortunately, this way we would have to do many changes in the code-base, so it seemed easier to go with the chosen approach.

3.4 Adding support for arbitrary number of 32-bit integer parameters

Adding support for arbitrary number of *i32* parameters was the next natural step to extend our wrapper. We explained what the caller frame slots are on the stack, and that the WebAssembly function expects the first 5 parameters to be in the proper registers (we will call these registers as *parameter registers*). If there are more than 5 parameters, then the remaining ones must be "properly" placed on the top of the stack. By properly we mean that the WebAssembly function expects that the converted

[1] *https://source.chromium.org/chromium/chromium/src/+/master:v8/src/wasm/wasm-objects.cc;l=1886;drc=953aea9232834912b2f01c8497bd87cb004cbe06?originalUrl=https:%2F%2Fcs.chromium.org%2F*

JavaScript parameters that did not fit into registers can be popped from the stack in an increasing order.

To maximize the code readability and avoid special case handling we decided to handle multiple *i32* parameters the following way[1]:

1. Before starting processing the parameters for the WebAssembly function call, we reserve 5 stack slots on the stack by decrementing the stack pointer.
   With this approach, we can always pop the values from the top 5 stack slots into the parameter registers before the WebAssembly function call.
   Note that if we have more parameters, we cannot move the parameter right after conversion into the proper parameter register, as the conversion functions that might be called for the other parameters can change the values inside these allocatable parameter registers.

2. We start looping through the JavaScript arguments from the caller frame slots and convert and move them onto the stack. As the WebAssembly function expects the parameters that did not fit into registers to be placed on the stack in a way it can pop them in an increasing order, we reserve on the stack the number of parameter slots, and place all the parameters after conversion into this area in proper order.
   Note that the order of processing the parameters is important. The reason is that the conversion of the JavaScript values to WebAssembly values is observable in JavaScript, e.g. the conversion may change the global state of JavaScript. Therefore, we have to convert the JavaScript arguments in an increasing order (e.g. starting with the first parameter).

3. Now all the parameters are converted and can be popped from the stack in an increasing order. Therefore, right before calling the WebAssembly function, we pop the top 5 values into the parameter registers.
   Note that we delete the spill slots after calling the WebAssembly function by incrementing the stack pointer.

The above mentioned parameter handling algorithm is illustrated in Figure 3.4. We can see the 5 reserved stack slots (colored with violet) under (in the direction of growth) the last spill slot, and the properly placed converted parameters that are popped to the parameter registers later.

[1] *https://chromium-review.googlesource.com/c/v8/v8/+/2381459*

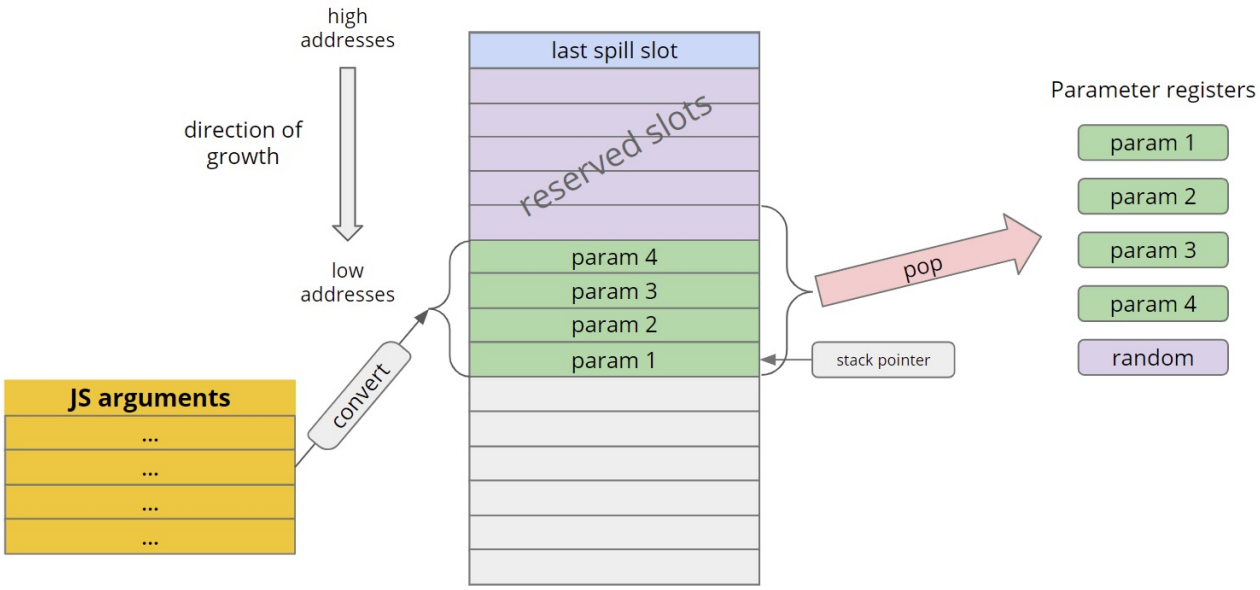

Figure 3.4: *Preparing the stack before calling the WebAssembly function with 4 parameters*

In the former version of our wrapper, when it could handle WebAssembly functions with up to one 32-bit integer parameter, we could save values on the top of the stack. However, to support functions with arbitrary number of parameters, the parameters that did not fit into registers are expected to be on the top of the stack before calling the WebAssembly function, which means that we cannot store other values on the top. Therefore, we had to change the GC support, as before we stored the pointers to the JavaScript heap objects on the top of the stack.
We decided to save a value that indicates how many values should be scanned after garbage collection. We could access this value and the values that have to be scanned relative to the frame pointer and iterate and update them.

We also decided to save onto the stack the information from the signature: number of parameters, number of return values, and the pointer to the array of parameter and return value types. It seemed easier to access them from the stack instead of loading them all over again.

In Figure 3.5, we can see the indicating value for the number of slots that need to be scanned after GC (*GC scan slot count*), and three slots containing the information from the signature (*parameter count, return count, ptr to the array of types*).

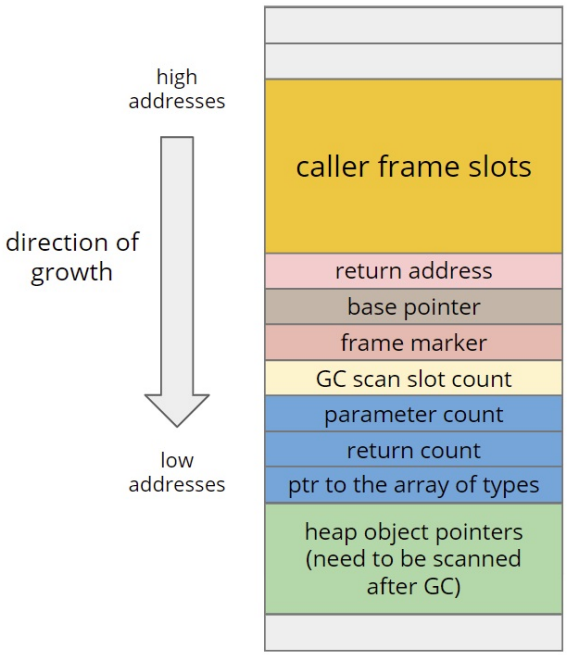

Figure 3.5: *Spill slots on the stack*

3.5 Adding support for 32-bit integer return value handling

As our next step, we decided to add support for *i32* return value in the generic wrapper[1]. We already showed in Figure 3.5 that the *return count* stack slot contains the number of return values.

If we have a return value, we have to convert it. After calling the WebAssembly function, we expect the return value to be in register *rax*. As we already mentioned in Section 2.2, pointer compression in V8 is set by default (but may be disabled). We learned that V8 uses tagged values, i.e. the least significant bit is used to distinguish *Smis* from heap object pointers. With compressed values, *Smis* can carry only a 31-bit payload. Therefore, we had to check if the value had fitted into 31-bits. If not, we converted it to a JavaScript type, a *HeapNumber*, by calling the proper built-in function:

```
1  Label to_heapnumber;
2  // If pointer compression is disabled, we can convert the return value to a Smi.
3  if (SmiValuesAre32Bits()) {
4    __ SmiTag(return_reg);
5  } else {
6    Register temp = rbx;
```

[1] *https://chromium-review.googlesource.com/c/v8/v8/+/2390141*

```
7       __ movq(temp, return_reg);
8       // Double the return value to test if it can be a Smi.
9       __ addl(temp, return_reg);
10      // If there was overflow, convert the return value to a HeapNumber.
11      __ j(overflow, &to_heapnumber);
12      // If there was no overflow, we can convert to Smi.
13      __ SmiTag(return_reg);
14    }
15    __ jmp(&return_done);
16
17    // Handle the conversion of the return value to HeapNumber when it cannot be a
18    // Smi.
19    __ bind(&to_heapnumber);
20    // We have to make sure that the kGCScanSlotCount is set correctly. For this
21    // builtin it's the same as for the Wasm call = 0, so we don't have to reset
22    // it.
23    __ Call(BUILTIN_CODE(masm->isolate(), WasmInt32ToHeapNumber),
24            RelocInfo::CODE_TARGET);
25
26    __ jmp(&return_done);
```

3.6 Adding support for other parameter and return value types

We want our wrapper to support the four standard WebAssembly types, i.e. *i32, i64, f32* and *f64*. Extending the wrapper to support arbitrary number of *i32* and *i64* parameters was quite straightforward. While converting parameters, we iterated through the array that contains the types of the parameters and the return values, and called the proper conversion function based on that type. Similarly, if the return value was *i64*, we called the built-in function to convert the value into the proper JavaScript value, a *BigInt*.

When dealing with floating-point values, we have to use different, floating-point registers. To add support for *f32* and *f64* return value, we just had to call the proper built-in function and use the floating-point registers. However, dealing with arbitrary number of all the four standard type parameters is a bit more complicated. In Subsection 3.1.1, we said that the WebAssembly function expects the first 5 integer and first 6 floating-point parameters in the proper "parameter registers". The function expects that the remaining parameters (i.e. those that did not fit into the parameter registers) can be popped from the stack in an increasing order.

First, we might think that we can just similarly convert and then place the parameters onto the top of the stack as before, and pop them to proper parameter registers based on the array that contains the value types. However, now that we have two types of parameter registers, popping or moving the values into them can create gaps on the stack and the remaining parameters will not be contiguously on the top of the stack (i.e. the WebAssembly function cannot pop them properly). In the next section, we describe the modified algorithm to avoid this problem.

The problem with the gaps on the stack after popping the converted parameters into the parameter registers is illustrated in Figure 3.6 (in this example there are only 3 general purpose and 3 floating-point parameter registers).

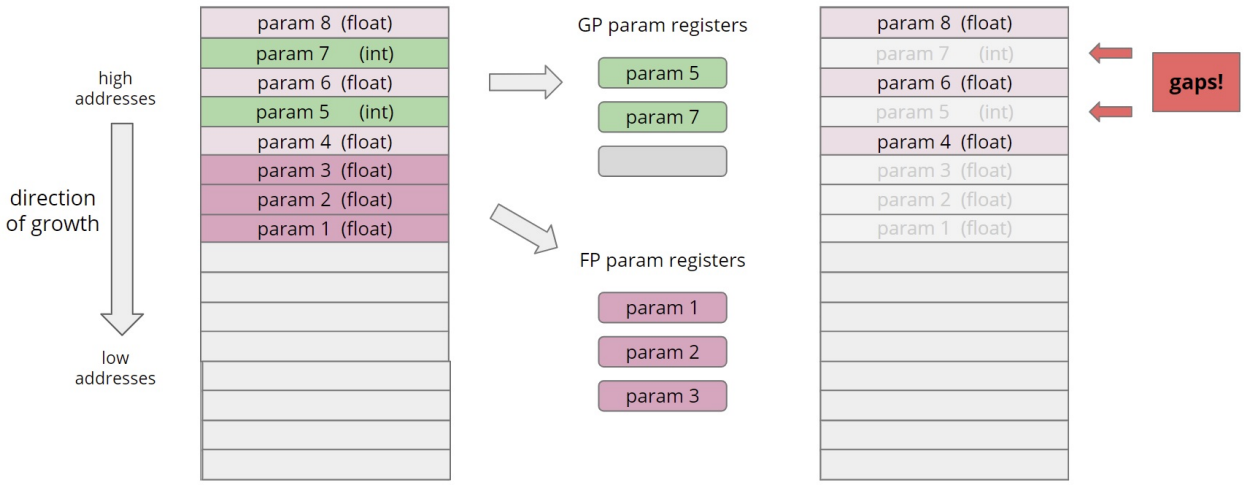

Figure 3.6: *Popping the converted integer and floating-point parameters from the stack similarly as in the former version of the generic wrapper*

3.7 Handling arbitrary number of integer and floating-point parameters

Our main goal was to efficiently prepare the parameters for the WebAssembly function in our wrapper. However, we wanted to avoid special case handling to make the assembly code easy to read. We decided to use the following algorithm[1]:

1. We create 2 sections on the stack for the converted parameters based on their types: *Integers* and *Floats*. This approach will help us when filling the parameter registers, because the types (integers: *i32* and *i64*, floats: *f32* and *f64*) will not be mixed up.

   We iterate through the caller frame slots to access the JavaScript arguments as before, however, this time after converting a value, we place it to the proper section. Both of the sections will have the size of the number of parameters.

2. After all the parameters are in their proper section, we move the top parameters from both sections into the proper parameter registers. We save two pointers into registers that point to the first parameters of the sections that have not been

[1] `https://chromium-review.googlesource.com/c/v8/v8/+/2429266`

moved to the parameter register yet. (Note that this time we will be moving the parameters and not popping.)

3. Next, we have to move the remaining parameters that did not fit into the parameter registers onto the top of the stack. We start iterating through the array that contains the types of the parameters from the back. Based on the pointers pointing to the remaining parameters of both of the sections, we place the remaining parameters onto the top of the stack. Going through the array containing the types in reversed order ensures that the parameters are placed in the correct order for the WebAssembly function.

Figure 3.7 illustrates the algorithm if there are only 3 general purpose and 3 floating-point parameter registers.

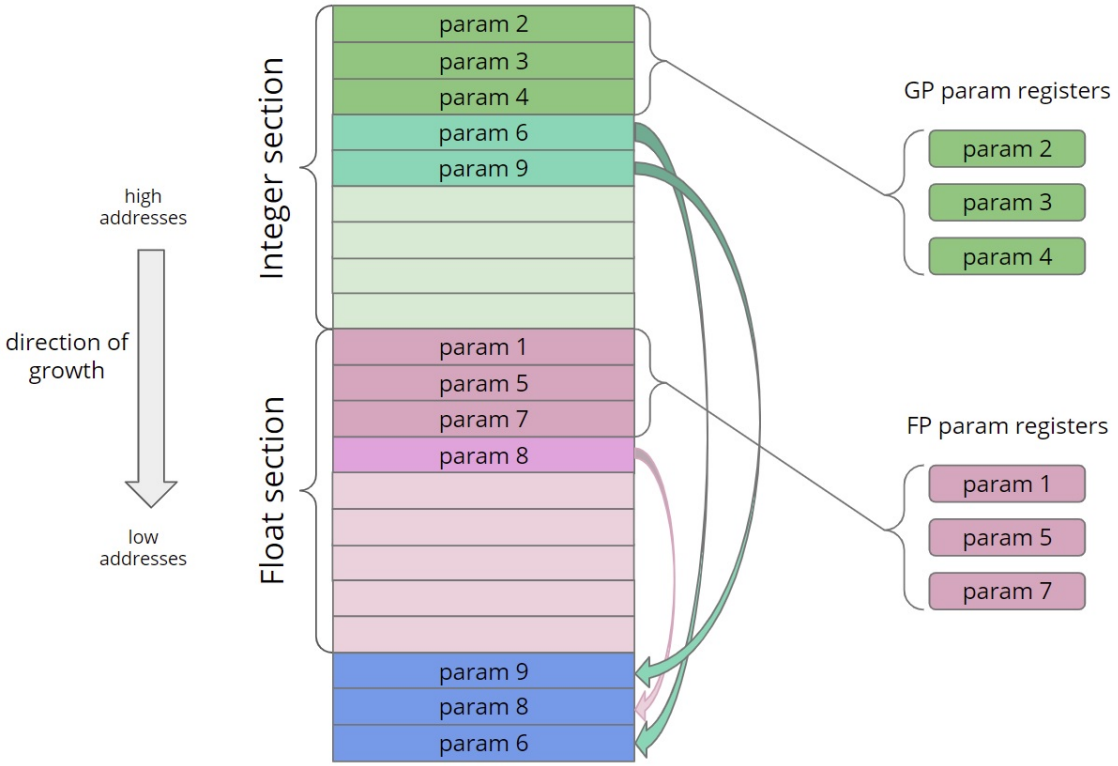

Figure 3.7: *Handling arbitrary number of integer and floating-point parameters in the generic wrapper*

Chapter 4

Results

We created a generic wrapper that can handle standard exported WebAssembly functions, i.e. exported functions with arbitrary number of standard WebAssembly type parameters, and zero or one standard WebAssembly type return value. In this chapter, we will show the results of measurements when using the generic wrapper function instead of the specific ones.

To use the generic wrapper, we have to set the flag that enables it. However, we would like to use the generic wrapper by default. A follow-up work enabled our wrapper by default, and also defined a strategy to compile specific wrappers after frequently calling WebAssembly functions with the same signature (we will refer to this follow-up work as a tier-up strategy). After the generic wrapper was enabled, we saw improvements in startup time that we will show below.

4.1 Compilation time and runtime

To measure the improvement in compilation time, we ran the Angry Bots game benchmark which consists of a *10 MB* large WebAssembly module. With the generic wrapper, we avoided the compilation of *159* specific wrapper.
In Figure 4.1, we can see that the compilation decreased by *14%* on average based on the Angry Bots benchmark.

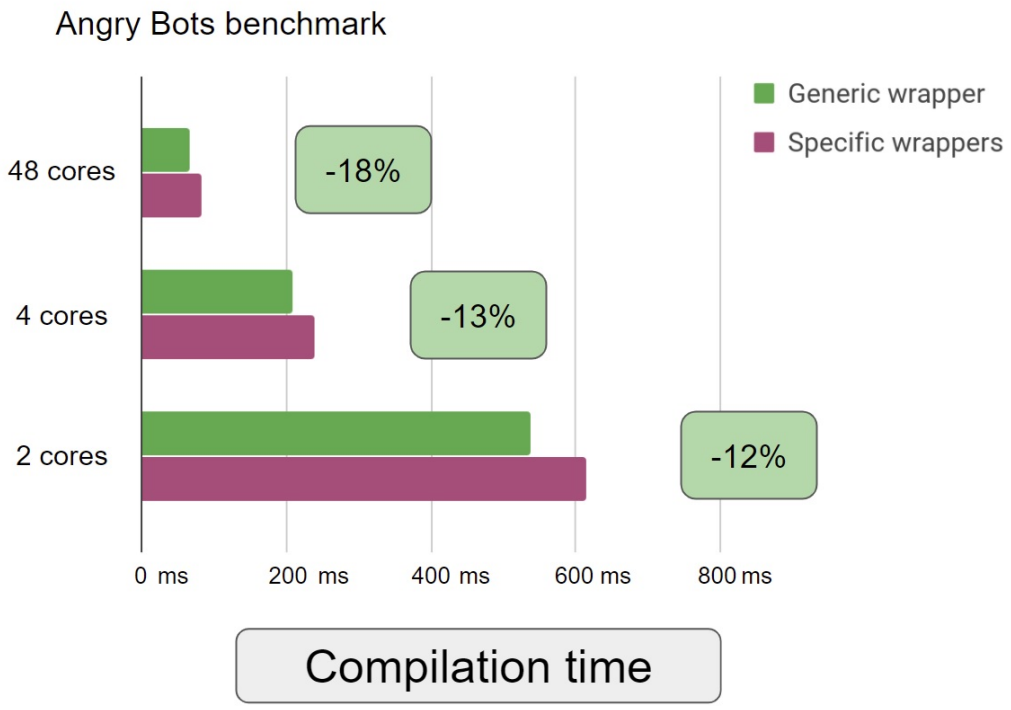

Figure 4.1: *Measuring compilation time with the Angry Bots benchmark. The benchmark was measured on 2, 4 and 48 cores.*
The average compilation time without using the generic wrapper on 2 cores is 614.12 ms, on 4 cores 238,66 ms and on 48 cores 81.93 ms. While using the generic wrapper, the compilation time decreased on 2 cores to 538.55 ms, on 4 cores to 207.80 ms and on 48 cores to 67.14 ms.

As the generic wrapper is more complex than the specific wrappers, we wanted to measure also the runtime cost of using the generic wrapper instead of the specific ones.

We used an internal function that creates and uses exported and imported wrappers for functions with 1, 2, 4 and 10 parameters of the four standard WebAssembly types, respectively. In Figure 4.2, we illustrated the growth in runtime due to the complexity of the generic wrapper.

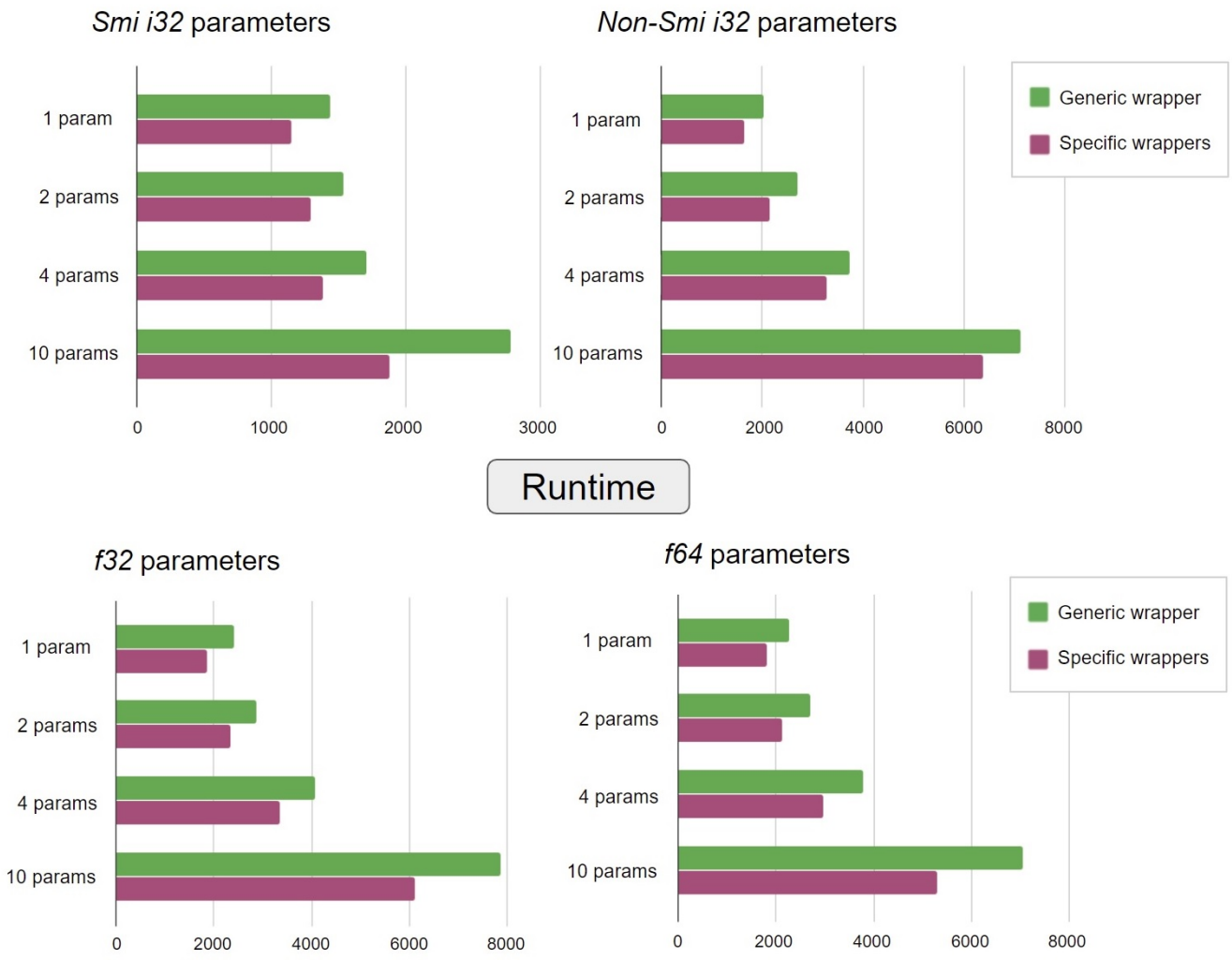

Figure 4.2: *Measuring runtime (in ms) with an internal function that creates and uses wrappers for exported and imported functions. We measured the regression in runtime separately for functions with 1, 2, 4 or 10 parameters.*
On average, the runtime increased for the Smi parameter functions by 28.8%, for the not Smi integer parameter functions by 18.5%, for f32 parameter functions by 27.5%, and for the f64 parameter functions by 25.3%.

4.2 Using the generic wrapper in production code

In the previous section, we saw that using the generic wrapper improves the compilation time, however, when used frequently instead of the specialized wrappers, it worsens runtime. After we had finished the implementation of our generic wrapper, a tier-up strategy was introduced to solve this issue. After a threshold of same signature function calls is reached, we compile the specific wrapper for the signature and use it instead of the generic one. The follow-up work also resulted in enabling the generic wrapper by default in *Google Chrome 89.0.4339.0* (March 2021).

We observed a huge improvement in deserialization time of Google Earth. For multi-threaded WebAssembly applications, the compiled WebAssembly code is shared across all threads. However, each thread needs its own set of export wrappers. The generic wrapper eliminated the need to re-compile the specific export wrappers, therefore, we reduced the deserialization time of Google Earth by **84%**. This is illustrated in Figure 4.3.

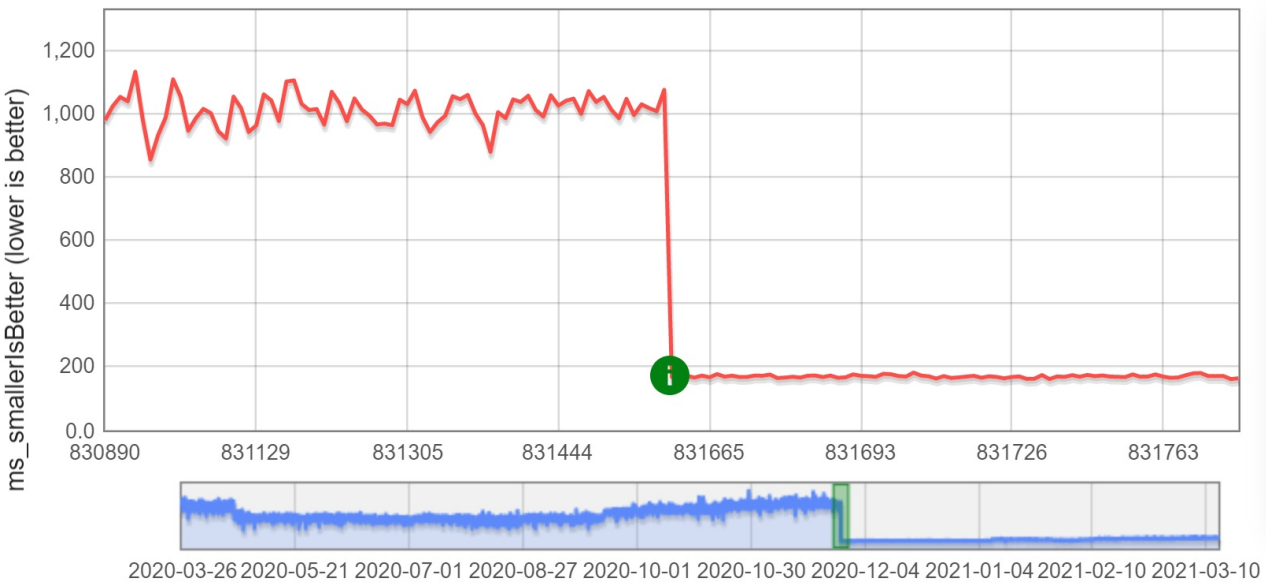

Figure 4.3: *Performance measurement of Google Earth's deserialization time. We can see significant improvement after using the generic wrapper. We reduced the deserialization time from 1074.88 ms to 168.101 ms. The whole graph is available on https://chromeperf.appspot.com/report?sid=624f9526e10146ecb3e4bcdf184be0ab5bc8a7df0561a8ad59a39777eae0de8d&rev=831654.*

Conclusion

In this work, we successfully designed and implemented a generic wrapper function for exported WebAssembly functions with arbitrary number of standard type parameters, and zero or one standard type return value.

First, we started with a brief explanation of the structure of web browsers to locate the JavaScript and WebAssembly engine. We explained the main concepts of WebAssembly, the compilation and interpretation phase in V8 and also described how the specialized wrapper functions were created in this engine.
The core part of this thesis explains our design approaches and the gradual process of extending our wrapper to support more and more WebAssembly functions. We had to overcome several challenges. We worked with a huge code-base and we had to get familiar with the V8 macro assembly language to write the body of our wrapper. After considering several options, we dealt with accessing the signature, added garbage collection support, and learned how to set up the stack for the WebAssembly functions. Some of the trickiest parts of the work were using the correct register operations to rewrite or to use the proper parts of memory locations, deal with pointer compression, reconsider what happens at compile time and runtime, and debugging our code. While implementing the algorithms, we always tried to create efficient code that can be read easily.
After we had finished the implementation, we made measurements that proved that our work was successful. We saw that after using the generic wrapper the compilation time and deserialization time was reduced. Therefore, we managed to improve the browsing experience for millions of people worldwide.

Obviously, the generic wrapper can be further extended to support even more WebAssembly functions, e.g. with multiple return values, imported WebAssembly functions, or JavaScript reference type parameters. Furthermore, we implemented our wrapper only for architecture *x64*, therefore, implementing it for all the other platforms will have to be done in the future.

Bibliography

[1] Cromium source code from Chromium Code Search – V8 JavaScript Engine. https://source.chromium.org/chromium/chromium/src/+/master:v8/. Accessed: 2021-02-07.

[2] WebAssembly. https://webassembly.org/. Accessed: 2020-11-16.

[3] Google Chrome Memory Usage – Good and Bad. https://blog.chromium.org/2008/09/, Sep 2008. Accessed: 2020-11-16.

[4] Experimental support for WebAssembly in V8. https://v8.dev/blog/webassembly-experimental, Mar 2016. Accessed: 2020-11-20.

[5] WebAssembly Specification. https://webassembly.github.io/spec/core/index.html#, 2020. Accessed: 2020-11-16.

[6] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullmann. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[7] Jesse Burstyn, Katricia Barleta, Lukas Berk, P Bennett Cole, and Tom Franzon. Conceptual Architecture of Google Chrome. Queen's University at Kingston, 2009.

[8] Lin Clark. A crash course in just-in-time (JIT) compilers – Mozilla Hacks – the Web developer blog. https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/, February 2017. Accessed: 2021-02-06.

[9] Lin Clark. Creating and working with WebAssembly modules – Mozilla Hacks – the Web developer blog. https://hacks.mozilla.org/2017/02/creating-and-working-with-webassembly-modules/, February 2017. Accessed: 2021-01-13.

[10] Cliff Click and Michael Paleczny. A simple graph-based intermediate representation. *ACM Sigplan Notices*, 30(3):35–49, 1995.

[11] Nick Fitzgerald. Oxidizing Source Maps with Rust and WebAssembly – Mozilla Hacks – the Web developer blog. `https://hacks.mozilla.org/2018/01/oxidizing-source-maps-with-rust-and-webassembly/`, January 2018. Accessed: 2021-01-14.

[12] Alan Grosskurth and Michael W Godfrey. A reference architecture for web browsers. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 661–664. IEEE, 2005.

[13] Franziska Hinkelmann. Franziska Hinkelmann: JavaScript engines – how do they even? | JSConf EU. `https://youtu.be/r5OWCtuKiAk`, 2016.

[14] Uday Hiwarale. The anatomy of WebAssembly: Writing your first WebAssembly module using C (C++). `https://medium.com/jspoint/the-anatomy-of-webassembly-writing-your-first-webassembly-module-using-c-c-d9ee18f7ac9b`, Dec 2020. Accessed: 2021-02-06.

[15] Shanhong Liu. Desktop internet browser market share 2015-2020. `https://www.statista.com/statistics/544400/market-share-of-internet-browsers-desktop/`, Oct 2020. Accessed: 2020-11-16.

[16] Ross McIlroy. BlinkOn 6 day 1 Talk 2: Ignition – an interpreter for V8. `https://youtu.be/r5OWCtuKiAk`, 2016.

[17] Ross McIlroy. Firing up the Ignition Interpreter. `https://v8.dev/blog/ignition-interpreter`, Aug 2016. Accessed: 2021-04-30.

[18] Mozilla Contributors. MDN Web Docs: WebAssembly. `https://developer.mozilla.org/en-US/docs/WebAssembly`. Accessed: 2020-11-16.

[19] Optasy. WebAssembly vs Javascript: Is WASM Faster than JS? When Does JavaScript perform Better? `https://medium.com/@OPTASY.com/webassembly-vs-javascript-is-wasm-faster-than-js-when-does-javascript-perform-better-db86d2ecf2cc`, Dec 2018. Accessed: 2021-02-13.

[20] Surma Surma and Deepti Gandluri. WebAssembly for Web Developers (Google I/O '19). `https://youtu.be/njt-Qzw0mVY`, 2019.

[21] V8 Team. V8. `https://v8.dev/`. Accessed: 2020-11-16.

Appendix

In the thesis, we extended V8, the JavaScript and WebAssembly engine that is available at *https://chromium.googlesource.com/v8/v8.git*.

To clone the repository, run the following command:

```
git clone https://chromium.googlesource.com/v8/v8
```

To view the changes created by us, run the following command:

```
git log --author="evih"
```

The source code written by us is also available at *https://chromium-review.googlesource.com/q/owner:evih%2540google.com+status:merged+before:2020-11-01*. We recommend viewing the source code on this website. The website contains a list of changelists (CLs) ordered by date. By clicking on a CL, you will see all the files that were modified in the CL. To see the exact changes in a file, click on the filename. In the thesis, we described the implementation of the generic wrapper based on the submission order of the CLs.

An alternative way for viewing the source code is to view it from the attached DVD. Please read the *README* file first to navigate through the files on the DVD.