

WebAssembly in Node.js

by

Tobias Nießen

Bachelor of Computer Science, Leibniz Universität Hannover, 2018

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF

Master of Computer Science

In the Graduate Academic Unit of Computer Science

Supervisors: Kenneth B. Kent, Ph.D.,
 Faculty of Computer Science

 Panos Patros, Ph.D.,
 Faculty of Computer Science

Examining Board: Gerhard Dueck, Ph.D.,
 Faculty of Computer Science, Chair

 Rainer Herpers, Ph.D.,
 Faculty of Computer Science

 Brent Petersen, Ph.D.,
 Faculty of Electrical and Computer Engineering

This thesis is accepted by the
Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

December, 2020

© Tobias Nießen, 2021

Abstract

Alongside JavaScript, V8 and Node.js have become essential components of contemporary web and cloud applications. With the addition of WebAssembly to the web, developers finally have a fast platform for performance-critical code. However, this addition also introduces new challenges to client and server applications. New application architectures, such as serverless computing, require instantaneous performance without long startup times. This thesis investigates use cases and integration issues of WebAssembly in Node.js, and the performance and quality of WebAssembly compilation in V8 and Node.js. We present the design and implementation of a multi-process shared code cache for Node.js applications, and demonstrate how such a cache can significantly increase application performance, and reduce application startup time, CPU usage, and memory footprint.

Acknowledgements

I would like to thank my supervisors Kenneth Kent from the University of New Brunswick and Panos Patros from the University of Waikato for their continuous support, Michael Dawson from IBM Canada for his valuable technical input, and Stephen MacKay from the University of New Brunswick for his unrelenting efforts to improve my writing skills and style.

This research was conducted within the Centre for Advanced Studies–Atlantic, Faculty of Computer Science, University of New Brunswick. The authors are grateful for the colleagues and facilities of CAS Atlantic in supporting our research. The authors would like to acknowledge the funding support of the Natural Sciences and Engineering Research Council of Canada (NSERC), 501197-16. Furthermore, we would also like to thank the New Brunswick Innovation Foundation for contributing to this project.

Table of Contents

Abstract	ii
Acknowledgments	iii
Table of Contents	iv
List of Tables	viii
List of Figures	ix
1 Introduction	1
2 Background and Related Work	5
2.1 From JavaScript to WebAssembly	5
2.2 WebAssembly Concepts and Terminology	9
2.2.1 Workflow	9
2.2.2 WebAssembly Modules	11
2.2.3 WebAssembly Instances	17
2.3 WebAssembly Performance	18
2.4 WebAssembly Security	20

2.5	WebAssembly and Serverless Computing	21
2.6	Related Code Caching Technologies	23
2.7	Research Outlook	24
3	Considerations for Using WebAssembly in Node.js	26
3.1	JavaScript and WebAssembly in Node.js	27
3.2	Node.js Native Addons versus WebAssembly	28
3.3	WebAssembly System Interface (WASI)	30
3.4	Use Cases of WebAssembly in Node.js	31
3.4.1	Side-channel Attacks Against Cryptographic Procedures	33
3.5	Integration into JavaScript Applications	34
3.5.1	WebAssembly JavaScript Interface	34
3.5.2	Data Access from WebAssembly	38
3.5.2.1	Example	38
3.5.2.2	Complex Data Structures	42
3.5.3	Memory Access from JavaScript	43
3.5.4	Reference Types	46
3.5.4.1	Using the Reflect Interface	47
3.5.5	Blocking versus Non-blocking Behavior	51
3.5.5.1	Communication Difficulties	52
3.5.5.2	Implementation of Synchronous Communication	54
3.6	Porting N-API to WebAssembly	57
3.6.1	Exception Handling	59
3.6.2	Pointers to JavaScript Values	60

3.6.3	Access to Memory Allocated by Node.js	62
3.6.4	Asynchronous Tasks	63
3.7	Summary	63
4	Compilation at Runtime	65
4.1	Compilation in the WebAssembly JavaScript Interface	67
4.2	Performance of Generated Code	69
4.2.1	Experimental Methodology	69
4.2.2	Results	71
4.3	Quality of Generated Code	73
4.3.1	Example	73
4.3.2	Example with Memory Access	76
4.3.3	PolyBench/C	81
4.3.3.1	Instruction Count and Branch Instructions	81
4.3.3.2	Memory Access	84
5	Compiled Code Caching	88
5.1	Code Extraction and Insertion	89
5.2	Compiler Performance versus Code Caching	93
5.3	Shared Code Cache	99
5.3.1	WebAssembly Module Identification	100
5.3.2	Design	103
5.3.2.1	Client Processes	104
5.3.2.2	Cache Server Process	105
5.3.2.3	Compiler Processes	105

5.3.3	Implementation	107
5.3.4	Evaluation	109
5.3.4.1	Evaluation using PolyBench/C	110
5.3.4.2	Evaluation using RustPython	112
5.4	Summary	121
6	Conclusion and Future Work	122
6.1	Threats to Validity	124
6.2	Future Work	124
	Bibliography	126
	Vita	

List of Tables

3.1	Qualitative comparison of JavaScript, WebAssembly, and native code in Node.js	32
3.2	Inter-thread communication model between synchronous and asynchronous tasks	56
4.1	Required process options for desired behaviors of synchronous WebAssembly module creation in V8	70
5.1	Mean compilation time of the RustPython interpreter in Node.js . . .	115
5.2	Total duration of WebAssembly compilation and Python code executions	120

List of Figures

2.1	WebAssembly workflow	10
2.2	Structure of a WebAssembly module	12
3.1	Basic communication with WebAssembly in a worker thread	53
4.1	Speedup of code generated by TurboFan with respect to Liftoff	71
4.2	Speedup of compiled code with respect to interpretation	72
4.3	Number of CPU instructions generated by each compiler for each of the PolyBench/C benchmarks	82
4.4	Number of CPU branch instructions generated by each compiler for each of the PolyBench/C benchmarks	82
4.5	Percentage of calls to error handlers with respect to all call instructions generated by each compiler for each of the PolyBench/C benchmarks	83
4.6	Number of CPU jump instructions generated by each compiler for each of the PolyBench/C benchmarks	83
4.7	Percentage of CPU instructions that access main memory	86
4.8	Percentage of CPU <code>mov</code> instructions that access main memory	86
5.1	WebAssembly cache data flow	90
5.2	Ratio of serialized compilation result size to WebAssembly module size	92

5.3	Ratio of serialized compilation result size to WebAssembly module size after applying <code>wasm-strip</code>	93
5.4	Compilation times by approach	95
5.5	Compilation CPU times by approach	95
5.6	Compilation memory usage by approach	96
5.7	Significance of compilation time improvements	96
5.8	Significance of compilation CPU time improvements	97
5.9	Significance of memory usage improvements	97
5.10	Performance comparison between <code>fp16</code> , <code>CRC32C</code> , and <code>SHA-1</code>	102
5.11	Shared cache server architecture	104
5.12	Cache design: Client process control flow	106
5.13	Cache design: Server process control flow	107
5.14	Cache design: Compiler process control flow	108
5.15	Shared cache performance impact on PolyBench/C benchmarks . . .	111
5.16	Overview of the use of the Python interpreter in the benchmarked Node.js application	113
5.17	Python execution time in tiering-up compilation mode	117
5.18	Cumulative compilation and execution time in selected scenarios . . .	119

Chapter 1

Introduction

WebAssembly is a new hardware abstraction for the web that aims to be faster than interpreted languages without sacrificing portability or security. Conceptually, WebAssembly is a virtual machine and binary-code format specification for a stack machine with separate, linearly addressable memory. However, unlike many virtual machines for high-level languages, the WebAssembly instruction set is closely related to actual instruction sets of modern processors, since the initial WebAssembly specification does not contain high-level concepts such as objects or garbage collection [15]. Because of the similarity of the WebAssembly instruction set to physical processor instruction sets, many existing “low-level” languages can already be compiled to WebAssembly, including C, C++, and Rust.

WebAssembly also features an interesting combination of security properties. By design, WebAssembly code can only interact with its host environment through an application-specific interface. There is no built-in concept of system calls, but they can be implemented through explicitly imported functions. This allows the host

to monitor and restrict all interaction between WebAssembly code and the host environment. Another important aspect is the concept of *linear memory*: Each WebAssembly instance can access memory through *linear memory*, a consecutive virtual address range that always begins at address zero. The host environment needs to translate virtual memory addresses into physical addresses on the host system, and ensure that virtual addresses do not exceed the allowed address range. On modern hardware, this can be implemented using the Memory Management Unit (MMU) and hardware memory protection features, leading to minimal overhead while allowing direct memory access to *linear memory* and preventing access to other memory segments of the host system [15]. Combined, these properties allow running the WebAssembly code both with full access to the real system, and in a completely isolated sandbox, without any changes to the code itself.

These properties make WebAssembly an attractive platform for performance-critical portable code, especially in web applications. However, WebAssembly makes no inherent assumptions about its host environment, and can be embedded in other contexts such as Node.js, a framework built on top of the JavaScript engine V8. Not only does Node.js share many technological aspects, such as the programming language JavaScript, with web applications, but its performance and portability goals align well with those of WebAssembly. Sandboxing, platform-independence, and high performance are especially relevant in cloud-based application backends, the primary domain of Node.js.

However, one hindrance remains: Because WebAssembly code is tailored towards a conceptual machine, it can either be interpreted on the host machine, or first compiled into code for the actual host architecture. As we will see below, interpretation

leads to inadequate performance. At the same time, compilation of large WebAssembly modules can lead to considerable delays during the application’s startup phase. Therefore, both strategies can result in performance issues.

Node.js is often used in *serverless computing*: Instead of keeping a number of servers running at all times, the provider allocates resources dynamically with a minimal scope. For example, Function-as-a-Service (FaaS), sometimes referred to as *serverless functions*, is a deployment model in which the provider only allocates enough resources for a single function to be executed for each request, and no internal state is preserved between requests [1]. In this scenario, it is crucial for the function to be executed quickly in order to produce the desired response to an incoming request with minimal latency, making it infeasible to compile large WebAssembly modules on each request.

This thesis analyzes the performance of V8’s WebAssembly compilers and performance improvements enabled by compiled code caching, and contributes:

- a detailed understanding of the difficulties of WebAssembly integration into Node.js applications;
- a comparison of the WebAssembly compilers within Node.js with respect to their effectiveness, meaning the quality of the compilation results, and their efficiency, meaning their performance and resource usage; and
- the design, implementation, and experimental evaluation of a multi-process shared code cache for WebAssembly code in Node.js applications.

Chapter 2 provides an overview of background and related work. In Chapter 3, we discuss various aspects of WebAssembly integration in Node.js applications, use cases, and challenges. We analyze the performance and quality of code generated

by V8 in Chapter 4. In Chapter 5, we investigate the performance of V8's WebAssembly compilers themselves, and the performance benefits of caching compiled code. We also present and evaluate the design and implementation of the previously mentioned multi-process shared code cache for WebAssembly code in Node.js applications. In Chapter 6, we draw conclusions from our work and provide future work for consideration.

Chapter 2

Background and Related Work

This chapter discusses existing technologies and works that are relevant to the research and ideas presented in this thesis.

2.1 From JavaScript to WebAssembly

JavaScript, or, formally, EcmaScript [13], is an object-oriented programming language, which was invented as a mechanism for adding dynamic behavior to web pages. Web pages can embed scripts written in JavaScript, which are executed by the user’s web browser when they visit the page. Interested readers are referred to “JavaScript: the first 20 years” by Wirfs-Brock et al. [75] for a detailed account of the language’s origin and development since 1995.

V8 [61] is a “JavaScript engine”. In other words, V8 is a software library that implements EcmaScript and that applications can use to execute JavaScript code. It was originally developed as part of the Chrome web browser, but a variety of other

programs are also using V8 today. One of these programs is Node.js [41], which was initially published by Ryan Dahl in 2009. It is a JavaScript runtime that can execute JavaScript code through V8 and provides a built-in software library (“core modules”) for JavaScript applications, as well as the ability to load additional software libraries. Unlike web browsers, Node.js does not contain features that are only meaningful in the context of web browsers, e.g., functions to interact with Document Object Model (DOM) objects on web pages. Instead, Node.js was designed for web server development [60], and the built-in software library contains classes and functions to facilitate networking using various protocols (including TCP, UDP, HTTP, HTTP/2, SSL/TLS, and DNS), file system access, compression, cryptography, and much more. Many of these features would be a security risk in web browsers, where web pages generally should not have direct access to critical resources, such as the system’s file systems. Node.js, on the other hand, was not designed to execute untrusted code. Starting with Node.js, JavaScript has become popular outside of web browsers, and, while Node.js was originally designed to implement web servers, it has seen immense growth in other areas, too. For example, Electron is a mature framework that allows building desktop applications using Node.js, and which has been used in a variety of successful, commercial products [40].

The immense success of JavaScript and Node.js has inspired a huge ecosystem of tooling and libraries. Software developers benefit from being able to use the same programming language, the same libraries, and the same tools for web pages, servers, command-line tools, desktop applications, etc. For web and cloud applications, Node.js enables developers to use the same technologies for front end and back end. Nonetheless, the use of JavaScript across different domains has also highlighted short-

comings of the language. While JavaScript has evolved considerably since its creation [75], it is still a scripting language whose performance often cannot compete with that of compiled code (see Section 2.3). In desktop or server contexts, it is often possible to mix slower, high-level languages with faster, low-level languages to boost performance. For example, Node.js allows embedding code written in C [26] or C++ [25] into JavaScript applications (see Section 3.2). However, this is generally not possible for JavaScript applications in web browsers.

To solve this problem without requiring new technologies, the language *asm.js* [20] was invented. This language is a subset of JavaScript and therefore works in any browser or other runtime that supports JavaScript. The subset was chosen such that it has the following properties:

1. It is possible to compile other languages, including low-level languages, such as C or C++, to *asm.js*.
2. It is possible to statically validate *asm.js* code, and to compile and optimize it ahead-of-time (AOT).

To facilitate the first property, *asm.js* supports a machine model that is similar to the machine model of low-level languages, such as C. For example, *asm.js* code emulates a heap memory section with byte addressing, in which C-style pointer operations can be performed. Similarly, even though functions are not stored in this heap, numeric function pointers can be used for indirect function calls.

To guarantee the second property, the *asm.js* specification defines a strict set of rules that can be used to check if JavaScript code is valid *asm.js* code. In particular, these rules enforce static type safety. The fact that JavaScript itself is a dynamically typed language makes ahead-of-time (AOT) compilation infeasible, and even just-in-

time (JIT) compilation difficult [47]. Unlike JavaScript, the asm.js subset guarantees static type safety to a degree that allows ahead-of-time compilation.

Thanks to this design and added support for ahead-of-time compilation of asm.js in major JavaScript implementations, including V8 [61], asm.js can offer considerable performance benefits over JavaScript code [20]. Tools such as Emscripten [76] can be used to compile code written in other languages, such as C, to asm.js code.

However, despite these performance advantages over JavaScript, asm.js is not an ideal solution. Startup performance is bounded by the fact that parsing JavaScript code, including asm.js code, is slow, especially since the generated asm.js code is often large. Additionally, many modern CPU features are unavailable in JavaScript. For example, JavaScript previously did not support 64-bit integer arithmetic, and even now the only way is to use potentially slow arbitrary-precision integers. Also, the performance of asm.js code vastly depends on the JavaScript implementation, whether it supports asm.js, and how it chooses to compile and optimize asm.js code [77].

These aspects illustrate the need for a new language that is not a subset of JavaScript. In 2017, WebAssembly 1.0 [66] was approved by representatives of companies and organizations that are responsible for the advancement of web technologies through their role in the development of web browsers and JavaScript implementations [72]. WebAssembly is not a programming language, but a hardware abstraction language: unlike JavaScript, it was not designed to be written by hand, but to be produced by compilers. Apart from not being a subset of JavaScript, WebAssembly has many similarities to asm.js, and aims to solve the previously mentioned problems while maintaining the qualities of asm.js. It is a low-level language that allows ahead-of-time compilation without sacrificing many of the security properties of high-level

languages (see Section 2.4), and allows WebAssembly code to use CPU features that would be unavailable to JavaScript code, e.g., fast 64-bit integer arithmetic. Additionally, parsing WebAssembly code is often much faster than parsing JavaScript or asm.js code, and the code size is generally smaller in the case of WebAssembly [77]. One downside of WebAssembly with respect to asm.js is the fact that support for the WebAssembly language must be provided by the runtime, whereas asm.js works correctly, but not necessarily with good performance, in any JavaScript runtime. Luckily, all major web browsers and Node.js support WebAssembly, meaning that software developers can already use the new technology to improve the performance of their applications [72].

2.2 WebAssembly Concepts and Terminology

This section attempts to clarify basic WebAssembly concepts and the terminology that the rest of this thesis will use when referring to the WebAssembly technology.

2.2.1 Workflow

WebAssembly code, similar to regular assembly code, is usually not written by hand, but produced by a compiler. Code written in programming languages such as C [26], C++ [25], Rust [35], C# [27], and many more can be compiled to WebAssembly. In most cases, existing compiler infrastructures, such as LLVM [31, 58], are used to produce an intermediate representation (IR) of the code that needs to be compiled. Only the final step, the conversion from intermediate representations to WebAssembly instruction sequences, is specific to the WebAssembly technology [7]. For example, the

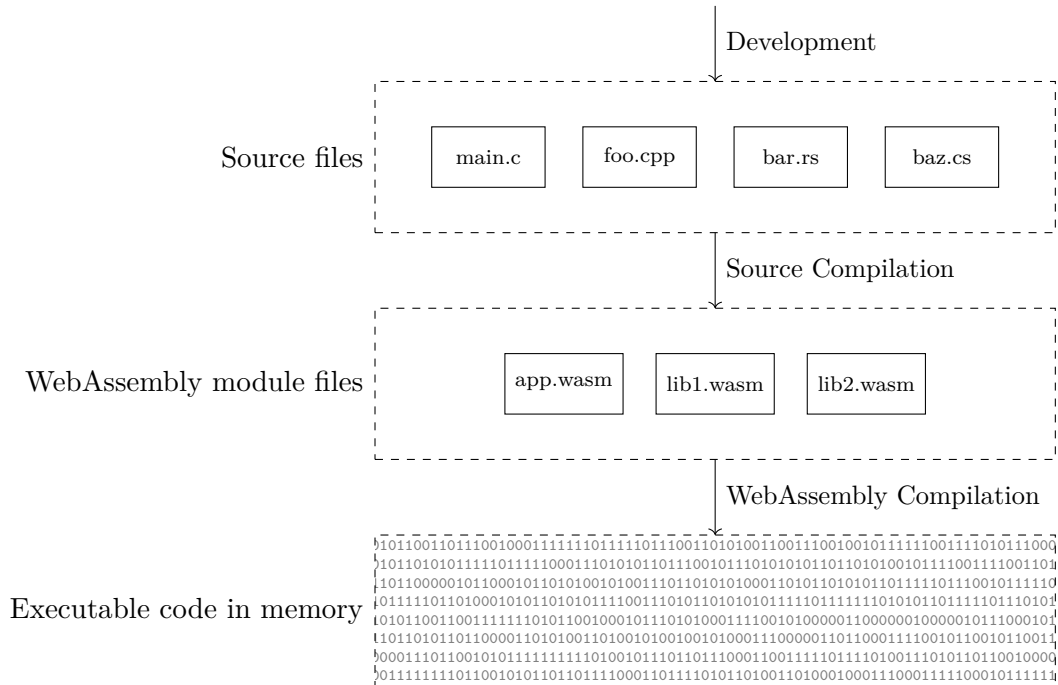


Figure 2.1: WebAssembly workflow

C compiler *Clang*, which is part of the LLVM project [58], can produce WebAssembly modules from C code. It is often used in combination with Emscripten [76], which provides implementations of the standard C and C++ libraries for WebAssembly, and, therefore, simplifies porting existing code to WebAssembly.

Unlike assembly code, WebAssembly code targets a conceptual machine, and cannot be executed by any (current) computer processor directly. In order to execute WebAssembly code, it must again be compiled to the target machine, or interpreted (see Figure 2.1). In this sense, WebAssembly is an intermediate code format.

This concept has been successful in the past. For example, Java bytecode, which is produced by compiling code written in the Java programming language, targets the conceptual Java Virtual Machine [46]. Java bytecode is stored in *class files*, which,

conceptually, are similar to WebAssembly modules. At runtime, class files, just like WebAssembly modules, must be compiled to the target architecture, or interpreted. However, Java and WebAssembly follow different goals: While Java bytecode was designed to support the Java programming language and requires parts of the Java standard library [46], WebAssembly is agnostic to the source language and does not require it to be object-oriented [15].

WebAssembly compilation at runtime will be covered in more detail in Chapter 4. While WebAssembly can be used “standalone”, e.g., using Wasmtime [6], it is more commonly used as part of larger applications. For example, while Node.js applications are mostly written in JavaScript, they may also make use of WebAssembly code. When a WebAssembly module is used as part of an application, and not as a standalone program, we call it “embedded,” and refer to the application as the “embedding application.”

Care must be taken not to mistake one compilation process with the other. We attempt to always unambiguously identify input and output languages when referring to compilation, for example, a “WebAssembly compiler” consumes WebAssembly code and outputs code for the target architecture, but it does not compile other languages or formats to WebAssembly.

2.2.2 WebAssembly Modules

A *WebAssembly module* is a collection of type, function, and data declarations and definitions. The structure is shown in Figure 2.2. Most sections are optional, and their order allows compilation in a single pass [15, 66].

Header
Type definitions
Import declarations
Function declarations
Table declarations
Linear memory declarations
Global variable declarations
Export declarations
Start function
Table definitions
Code section
Data section

Figure 2.2: Structure of a WebAssembly module

WebAssembly modules are typically generated by a compiler, e.g., a C, C++, or Rust compiler. The WebAssembly specification [66] defines a binary format for WebAssembly modules, which is used in practice, and a textual format, which is similar to assembly code. Where example modules, or parts thereof, are presented in this thesis, they will be presented in textual form. When referring to the structure or size of modules, the binary format is implied.

WebAssembly supports four scalar *value types*: 32-bit integers (`i32`), 64-bit integers (`i64`), 32-bit floats (`f32`), and 64-bit floats (`f64`). The type definition section allows

specifying function types as sequences of parameter types and sequences of return value types.¹ For example, the type of a function that accepts two 32-bit integer parameters and returns a single 32-bit integer value would look like this:

```
(type (func (param i32 i32) (result i32)))
```

WebAssembly functions consist of a function type and a sequence of WebAssembly instructions. These instructions operate on a conceptual stack machine, with the stack being initially empty. For example, the following instruction sequence loads the value of a local variable with index zero, adds a constant value of 0x1000, and assigns the result to another local variable.

```
local.get 0  
i32.const 0x1000  
i32.add  
local.set 1
```

The WebAssembly text format allows nesting instructions according to their operation on the stack in an almost functional manner. This does not affect the binary representation of the instruction sequence, and is merely a matter of presentation. The above instruction sequence is equivalent to this nested instruction:

```
(local.set 1 (i32.add (local.get 0) (i32.const 0x1000)))
```

Unlike traditional assembly languages, WebAssembly does not have a concept of a finite set of registers. Instead, functions can declare any number of local variables. Similar to local variables in programming languages, it is the WebAssembly

¹The initial WebAssembly specification [66] only allowed single return values. This restriction has since been lifted.

compiler's responsibility to assign local variables to registers or memory within the function's stack frame. Additionally, modules can declare global variables to store values of the above types. All variables and memory sections are initialized with the value zero, unless explicitly declared otherwise, meaning that the values of variables are never uninitialized or undefined in WebAssembly.

Apart from explicit access to local variables, assuming they are stored on the stack and not in registers, WebAssembly code cannot access its own stack frames. In fact, WebAssembly can only access a special part of the host system's main memory, which is referred to as *linear memory*. From the perspective of WebAssembly code, linear memory is a continuous address range beginning at address zero, and extending for a number of pages. All addresses within this range are valid, and all addresses outside of this range are invalid.² WebAssembly pages have a fixed size of 64 KiB, and linear memory can be grown dynamically using the `memory.grow` instruction. For example, assuming that the module's linear memory currently consists of ten pages, i.e., the valid address range is $[0..(10 * 65536 - 1)]$, the following instruction allocates five more pages, and, therefore, extends the valid address range to $[0..(15 * 65536 - 1)]$:

```
(memory.grow (i32.const 5))
```

Linear memory is strictly separated from global and local variables, functions, etc. This means that variables and functions, unlike in C and many other programming languages, do not have memory addresses in WebAssembly. Compilers that produce WebAssembly code can decide to place variables in linear memory to allow pointer arithmetic on them. However, this is not possible for functions, but function pointers are an essential part of most languages. Instead, WebAssembly provides *tables*,

²Since addresses are unsigned, any address is either within the range or too large.

which map indices to WebAssembly functions. The `call_indirect` instruction allows calling a function by its index in a table. This enables languages, such as C [26], to hide the fact that function pointers do not point to a location in memory, but rather to an entry in a table.

Apart from these restrictions, WebAssembly modules can use linear memory arbitrarily. Some languages might require pointer arithmetic within or across stack frames. In these cases, compilers that target WebAssembly can generate code to reserve a virtual address range within linear memory for the stack (“unmanaged stack”). Global variables can be used as a replacement for the stack pointer and base pointer registers [32]. Similarly, most programs utilize dynamic heap memory allocations, e.g., using `malloc` in C. Again, the compiler can generate WebAssembly code that implements memory management within linear memory. For example, WebAssembly itself does not provide high-level memory management such as `malloc`, but this does not prevent WebAssembly modules from defining their own `malloc` implementation. WebAssembly does not support arbitrary jump instructions, instead, its instructions resemble control flow in programming languages. For example, this instruction sequence sets `$i` to 10 if its current value is zero:

```
(if (i32.eqz (local.get $i))
  (local.set $i (i32.const 10)))
```

Similarly, the `loop` instruction can be used to create an instruction sequence in which the `br_if` instruction can be used to conditionally jump back to the beginning of the sequence. The following sequence is roughly equivalent to a C-style loop of the form `do { foo(i); } while (i-- != 0)`:

```
(loop $repeat
  (call $foo (local.get $i))
  (local.set $i (i32.sub (local.get $i) (i32.const 1)))
  (br_if $repeat (i32.ne (local.get $i) (i32.const 0))))
```

Finally, the `block` instruction works very similarly to the `loop` instruction, except that `br_if` and other branch instructions jump to the end of the block, not to its beginning. The `block` instruction, therefore, allows skipping subsequent instructions. Dynamic jumps through the `br_table` instruction allow efficient implementations of constructs such as C-style switch statements.

These structured control flow instructions serve two purposes. On one hand, they are close enough to low-level jump instructions to make WebAssembly compilation simple. On the other hand, they enforce enough structure to allow static validation of the code. For example, the instructions within a loop may not access data that is already on the stack, and any data added to the stack within the loop is discarded at the end of each loop iteration. Such rules allow static analysis of the stack height and data types at any point within a function, resulting in WebAssembly being type safe, with the exception of linear memory accesses.

The only way for WebAssembly code to interact with its environment is via imports and exports. Import declarations can be used to obtain references to functions provided by the host system. For example, WebAssembly code itself cannot access the user's terminal or the process's standard input/output streams, and therefore cannot display a message to the user. However, the embedding application might provide a function to display such a message, and the WebAssembly module can import it.

Analogous to import declarations, export declarations specify which declared entities (i.e., functions, linear memory, global variables, tables) should be exposed to the module's environment.

2.2.3 WebAssembly Instances

A WebAssembly module only describes the behavior and structure of the binary. A *WebAssembly instance* is a specific instance of the module and shares no state with other instances of the same module. Analogous to the relation between classes and objects in object-oriented languages, the module acts as a blueprint of the behavior and structure of WebAssembly instances.

Similarly, a WebAssembly module defines a set of *imports*, e.g., functions. However, these only define the type of what is to be imported, and not the implementation. A WebAssembly instance maps each import to an implementation, which must match the parameter and return value types defined in the respective WebAssembly module. This adds to the ability to isolate WebAssembly code from the host system. Since WebAssembly does not have a concept of traditional system calls via interrupts, access to the underlying host system can only be achieved through calls to imported functions (see Section 3.3). The host system can implement these functions arbitrarily to grant, restrict, or emulate access, without having to change the WebAssembly module, which only defines parameter and return values of these functions. In fact, the same process can use multiple instances of the same WebAssembly module, and these instances can have different levels of access to the host system.

2.3 WebAssembly Performance

Most existing research around WebAssembly focuses on performance comparisons between JavaScript, WebAssembly, and native code.

Haas et al. described the motivation behind WebAssembly, its design goals, code execution and validation [15]. The authors used the PolyBench/C benchmark [51] to compare the performance of WebAssembly to that of native code and asm.js [20] (see Section 2.1). They found that WebAssembly was 33.7% faster on average than asm.js, and that the execution time of WebAssembly was less than 150% of the native execution time for 20 out of 24 benchmarks. It is important to note that V8 only implemented the TurboFan compiler [62] at that time, and did not use the Liftoff compiler [18] that is discussed later in this thesis.

Herrera et al. used the Ostrich benchmark suite [29] to compare the performance of native code, WebAssembly, and JavaScript. The benchmark performs numerical computations that are deemed relevant for scientific computations, such as machine learning. While they also found WebAssembly in Node.js to be slower than native code, it consistently outperformed JavaScript in all environments [21, 22].

Malle et al. conducted experiments comparing WebAssembly to JavaScript, asm.js, and native code in the context of artificial intelligence algorithms [34]. The results are in line with the results reported by Haas et al. [15] and Herrera et al. [21, 22], and again show that WebAssembly is faster than JavaScript for scientific computations, but slower than native code. The authors suggest that future additions to WebAssembly such as *single instruction, multiple data* (SIMD) instructions will likely reduce the difference between WebAssembly and native code.

Matsuo et al. suggested using WebAssembly in browser-based volunteer computing. They found WebAssembly outperformed JavaScript for long-running tasks, but the overhead of compiling and optimizing WebAssembly before being able to run it caused performance gains to disappear for tasks with short durations [36].

Jangda et al. exposed performance flaws of WebAssembly implementations in web browsers [28]. In their research, they used the SPEC CPU[®] 2006 and SPEC CPU[®] 2017 benchmark suites³ to achieve more realistic results than previous publications that were focused on purely scientific computation, and found that, on average, WebAssembly code in the V8-based web browser Chrome was 55% slower than native code. To explain the performance gap, the authors analyzed the differences between compiled WebAssembly code generated by V8 and native code generated by a C compiler, and found multiple problems: In general, V8 produces more instructions, for example, because it does not utilize all available memory addressing modes and therefore needs to generate more instructions to calculate memory addresses. V8 also generates more branch instructions than the respective C compiler. The increased code size not only leads to more CPU cycles required to execute the code, it also causes more cache misses in the processor’s L1 instruction cache. Additionally, the code generated by V8 suffers from increased register pressure due to sub-optimal register allocations and the fact that some registers are reserved for memory management by V8. Finally, the WebAssembly specification mandates certain safety checks at runtime, for example, during dynamic branch or call instructions, which also incur a performance cost. However, despite these problems, the authors also showed that WebAssembly was 54% faster than asm.js in the same browser.

³SPEC is a registered trademark of the Standard Performance Evaluation Corporation.

Hu et al. demonstrated that the performance of WebAssembly is sufficient to perform speech recognition in real-time, for example, as part of a web application [23]. Similarly, Zhuravleva implemented live video processing, including object recognition in real time, in WebAssembly [79]. These examples show that WebAssembly has enormous potential to bridge the gap between fast low-level languages and high-level languages, such as JavaScript, in security-critical environments.

The multitude of publications highlighting the performance benefits of WebAssembly over JavaScript has inspired efforts to simplify the integration of WebAssembly into existing JavaScript applications. For example, Reiser et al. proposed a cross-compilation method from JavaScript to WebAssembly that resulted in significant speedups of computationally intensive algorithms [53]. Closely related are variants of JavaScript that target WebAssembly, such as AssemblyScript, which provides better performance than JavaScript [64]. Cabrera Arteaga et al. applied superoptimization to WebAssembly, and were able to improve WebAssembly code beyond optimizations provided by compilers [7].

2.4 WebAssembly Security

As discussed in the previous sections, WebAssembly provides extensive memory isolation features. Combined with the host system’s ability to intercept, deny, or emulate all accesses from WebAssembly code to its host system, or from the host system to WebAssembly, these security properties ensure that even vulnerable or faulty WebAssembly cannot escape isolation [16]. Many attacks, such as privilege escalation, are unlikely with these security guarantees.

In fact, these properties are so unique, that they have motivated research into re-structured operating systems. Wen et al. propose a new operating system that can execute WebAssembly code natively within the kernel [74]. According to their research, WebAssembly does not require the usual security restrictions imposed by an operating system distinguishing between kernel and user mode, because, by design, WebAssembly cannot access foreign resources.

Despite these concepts, WebAssembly does not automatically prevent many traditional memory-related vulnerabilities and attacks, as demonstrated by Lehmann et al. [32]. For example, if C code contains a vulnerability that allows an attacker to access or overwrite data in the program’s heap memory area, compiling the code to WebAssembly will likely not eliminate the vulnerability. However, an attacker would only be able to access the parts of the heap that belong to the vulnerable WebAssembly module, that is, its linear memory.

To mitigate such risks further, Disselkoen et al. proposed a memory-safe variant of WebAssembly, which requires compilers to add information about memory semantics to WebAssembly modules, which can then be used by a WebAssembly compiler or runtime to verify and enforce these semantics [12]. The authors criticize that WebAssembly was not designed with a greater focus on memory safety.

2.5 WebAssembly and Serverless Computing

Serverless Computing refers to on-demand resource provisioning, in which application instances can be created and destroyed at any point. When an application requires more resources, more instances of the same application are spawned with very

little delay, but on the other hand, when it has more resources than it needs, some of its instances might be destroyed. One such serverless architecture was invented by Varda et al., and allows running various instances of JavaScript and WebAssembly applications in a distributed cloud network [63]. Their concept focuses on executing untrusted third-party code in a secure and efficient manner.

Function-as-a-Service (FaaS), sometimes referred to as “serverless functions,” is a serverless deployment model in which the provider only allocates enough resources for a single function to be executed for each request, and no internal state is preserved between requests [1]. A well-designed application benefits from the scalability of this approach, meaning its ability to immediately adapt to changing workloads. In this scenario, it is crucial for the function to be executed quickly in order to produce the desired response to an incoming request with minimal latency.

Hall et al. investigated WebAssembly as a platform for serverless applications. They came to the conclusion that the security properties of WebAssembly allow isolation similar to virtualization via containers, and that, while WebAssembly generally did not outperform native code, containers often took longer to start than the respective WebAssembly implementations. This “cold start penalty” caused WebAssembly to provide better performance in certain scenarios [16].

Gadepalli et al. also proposed using WebAssembly as a basis for serverless computing. They also found that WebAssembly was a promising alternative to containers and virtual machines and allowed faster startup of applications, resulting in a lower initial latency [14].

Napieralla considered WebAssembly for edge computing on IoT devices and compared it to container-based solutions [37]. In line with the previous two publications,

they also found that WebAssembly provided fast startup times compared to containers, but did not outperform native solutions for long-running tasks.

One problem in serverless computing is that, since resources are provisioned on-demand, persistent data must be stored externally, and not within an instance of the application. This is especially relevant in FaaS models, where each function is supposed to be stateless, and can only persist state externally. Zhang et al. presented an approach to reduce data transmissions between computational units and storage nodes [78]. Instead of retrieving data and then performing computation on it, their design allows applications to provide WebAssembly code to storage nodes (“storage functions”), which then execute the given code on the data directly, without the need of transferring the data to the computational unit first.

2.6 Related Code Caching Technologies

The idea of caching compiled code beyond single processes is not new, and has been implemented for other languages. Bhattacharya et al. discussed improvements for the shared class cache (SCC) used by the J9 Java virtual machine. While its primary purpose is to reduce memory usage by sharing required class files, the SCC also contains compiled code, reducing application startup times significantly [2, 19]. Patros et al. invented a mechanism to reuse compilation results for Platform as a Service (PaaS) clouds via Dynamically Compiled Artifact Sharing (DCAS), with a focus on the Java SCC [48, 11].

Park et al. proposed a method to reuse code generated by an optimizing JavaScript just-in-time (JIT) compiler, allowing ahead-of-time (AOT) compilation based on

previous compilations of the same code. Their benchmarks demonstrated significant speedups in JavaScript application performance [47]. The authors also highlighted the need for such technologies due to the increasing code size of web applications. Haas et al. discuss two ways to improve compilation and startup times for WebAssembly in web browsers [15]. According to their research, parallel compilation using multiple threads leads to compilation times that are 80% lower than those of single-threaded compilation. V8 already implements parallel compilation by assigning individual WebAssembly functions to separate compilation threads. The authors also suggest that developers cache compiled WebAssembly modules in browsers using client-side IndexedDB databases [65].

However, IndexedDB is not available in Node.js, and as of 2020, support for WebAssembly modules in IndexedDB databases has been removed from V8, making it impossible for developers to explicitly cache compiled WebAssembly modules. Instead, web browsers are encouraged to implement implicit code caching as part of streaming WebAssembly compilation [4], which is not available in Node.js.

2.7 Research Outlook

The literature review discussed in the previous sections has inspired multiple research goals. First, while many existing publications analyze the performance of WebAssembly code and compare it to that of JavaScript and native code, they only provide limited insight into the potential of WebAssembly in the context of Node.js. For modern applications, performance is not the only critical factor during development and deployment. Chapter 3, therefore, considers other relevant aspects,

including interfaces between JavaScript, WebAssembly, and native code, and the interoperability of these technologies.

More importantly, while existing work compares the performance of WebAssembly to that of other technologies, it does not sufficiently distinguish between different WebAssembly compilation strategies in V8 and Node.js. Recent versions of V8 and Node.js provide multiple WebAssembly compilers with different performance goals and properties. Chapter 4 analyzes the performance and quality of code produced by the available WebAssembly compilers and briefly compares it to interpretation, which is another WebAssembly execution strategy provided by V8 and Node.js that does not require compilation.

Finally, existing research has not focused on the performance of WebAssembly compilation itself. Therefore, Chapter 5 discusses performance differences between different WebAssembly compilers and compilation strategies in V8 and Node.js, and proposes compiled code caching as a solution to long compilation times. While this solution takes inspiration from related work presented in Section 2.6, existing code caching solutions are either restricted to web browsers or were designed for other languages. Unlike these previous designs and implementations, our cache system is specific to WebAssembly and Node.js, even though it could be extended to other runtimes that support WebAssembly.

The idea of compiled code caching is closely related to the research referred to in Section 2.5, which shows that WebAssembly has the potential to be used in serverless computing, which often requires short application startup delays [1], and compilation at runtime can be a significant hindrance to that goal. Therefore, compiled code caching could improve WebAssembly’s suitability for serverless environments.

Chapter 3

Considerations for Using WebAssembly in Node.js

This chapter discusses aspects of Node.js and WebAssembly, and the relation between these two technologies. Just like Node.js itself, WebAssembly is not always an ideal choice, and many factors must be examined when considering its use.

We will consider three kinds of code that can be used within Node.js: JavaScript, native addons, and WebAssembly. This chapter will not focus on the pure performance of each kind of code, as the performance differences have already been explored at length by previous publications (see Section 2.3). All studies indicate that WebAssembly code is generally slower than native code, but can be faster than JavaScript. Similarly, we will not consider asm.js (see Section 2.1) from this point forward, since it has been effectively superseded by WebAssembly.

This chapter contributes a detailed understanding of the possibilities and hindrances of WebAssembly integration in Node.js applications based on existing language and

interface specifications, and combines information from these documents with experience in and knowledge about Node.js software development. Section 3.5.2 and Section 3.5.3 explore problems related to the data exchange between JavaScript and WebAssembly code. Section 3.5.5.2 introduces an open-source software library that was published by the author of this thesis to improve WebAssembly integration in Node.js applications.

3.1 JavaScript and WebAssembly in Node.js

JavaScript, or, formally, ECMAScript [13], is the primary programming language of the web and Node.js applications. JavaScript in web browsers is often referred to as “client-side JavaScript”, and JavaScript within Node.js as “server-side JavaScript”, since Node.js was designed to implement web servers [41]. The ability to use the same language for client-side and server-side code, potentially as part of the same application, has affected software development over the last decade [75]. For example, not only do developers not need to learn different programming languages for different parts of an application, but they can often use the same software libraries in browsers and in Node.js, which reduces the need to familiarize themselves with new application programming interfaces (APIs) and libraries. Similarly, many software development tools can be used both for client-side and server-side code, including test frameworks, integrated development environments (IDEs), and software debugging, monitoring, and profiling tools. Additionally, since JavaScript was designed to be portable and makes no assumptions about the underlying hardware or operating system, Node.js applications also benefit from portability and hardware-independence. All these

aspects equally apply to WebAssembly. While primarily designed for use in web browsers, it can also be integrated into Node.js applications. WebAssembly was designed to work well when embedded into JavaScript applications, which is the case both in browsers and in Node.js. Given how successful JavaScript has been on both sides [75], it is likely that WebAssembly will see wide adoption as well.

The *WebAssembly JavaScript Interface* allows JavaScript applications to interact with WebAssembly code and will be described in detail in Section 3.5.1.

3.2 Node.js Native Addons versus WebAssembly

With the addition of WebAssembly to Node.js, there are three kinds of code that are supported: JavaScript, native addons, and WebAssembly. While JavaScript code and WebAssembly are interpreted and/or compiled by V8 (see Chapter 4), native addons behave like shared libraries and allow embedding “native” code, e.g., compiled C or C++ code.

Unlike WebAssembly, native addons have direct access to the underlying system and its resources (file systems, network interfaces, etc.), with the only restrictions being imposed by the operating system. While this might be a desired or even required feature for some use cases, it might be a security risk in others [42].

In cloud server (or serverless) environments, native code requires stricter isolation than JavaScript and WebAssembly (see Section 2.5). For example, JavaScript and WebAssembly cannot perform system calls directly (see Section 3.3), and access to physical resources can thus be controlled by the JavaScript and WebAssembly runtime. Native code, on the other hand, must be isolated, e.g., by running inside of

a virtual container, to prevent it from accessing physical resources, and from using system calls to communicate with the operating system’s kernel.

Additionally, native addons usually need to be compiled on the target platform, while WebAssembly is portable and agnostic of the system architecture. WebAssembly can be deployed just like JavaScript files, and will work on any hardware and operating system. Native code, on the other hand, targets a specific hardware architecture and operating system. Installing software libraries or applications that use native addons thus usually requires compiling the source files into native code, which requires the target machine to have a compiler toolchain installed and configured properly.

Section 3.5.1 describes how WebAssembly functions can be called from JavaScript directly (and vice versa), eliminating the need for “glue code” in WebAssembly, and in the low-level code that was compiled to WebAssembly. This is not the case for native addons. Native functions cannot be called from JavaScript directly, meaning that C/C++ “glue code” is required to manage the interaction between JavaScript and native functions, to convert between JavaScript and native types, etc. For this purpose, native addons can use the *N-API* software library provided by Node.js [43], which consists of C type and function definitions that allow native C or C++ code to interact with Node.js and V8 and to consume and create JavaScript values. As Section 3.6 will show, compiling C or C++ code that uses N-API to WebAssembly is difficult, and might not be beneficial. Section 3.5.4 describes an alternative for interaction between JavaScript and WebAssembly beyond what is implicitly allowed by the WebAssembly JavaScript Interface (see Section 3.5.1).

As discussed in Section 2.3, current WebAssembly implementations are still slower than native code for many use cases, and their access to CPU features and hardware

acceleration is more restricted. Performance-critical code might therefore still benefit from the advantages of native code, at the cost of security and portability.

3.3 WebAssembly System Interface (WASI)

The *WebAssembly System Interface* (WASI) [5] was designed as a replacement for system calls (“syscalls”). Usually, low-level languages invoke system calls by intentionally causing a software interrupt, which allows the system’s kernel to handle the interrupt and execute the requested function.

WebAssembly does not have access to system calls for multiple reasons, one of them being that WebAssembly does not support interrupts. Additionally, system calls would allow WebAssembly to access the host system, which violates the security goals of WebAssembly. Finally, the mechanics of system calls depend on the operating system, and different operating systems provide different sets of system calls. WebAssembly, on the other hand, was designed to be portable, and to not depend on the underlying operating system.

However, most C and C++ applications make use of the standard C/C++ libraries (*libc* [26] and *libc++* [25], respectively), which include various functions that require system calls, e.g., file input/output operations, retrieving the current system time, etc. To solve this problem, the WebAssembly System Interface [5] is currently being designed, and an experimental implementation is available in Node.js [44]. Instead of platform-specific system calls, WASI consists of a set of platform-independent functions that can be imported by WebAssembly modules. These functions are sufficient to implement the C/C++ standard libraries on top of them.

This approach aligns well with the goals of WebAssembly. It preserves portability since the WASI specification describes the requirements for each “system call” regardless of the underlying platform. At the same time, since WASI “system calls” are regular functions, the host system can implement them arbitrarily, or even provide different WASI implementations for different instances of the same module.

Combined with a compatible standard library implementation, such as those provided by LLVM [31, 58] or Emscripten [76], WASI allows compiling existing C and C++ applications and libraries in a platform-independent manner, which can then be used on any platform that implements WASI. This is not limited to JavaScript runtimes such as Node.js and Deno [10], web pages can also provide their own JavaScript WASI implementations that simulate file systems and other system resources. Web browsers could allow users to selectively grant WebAssembly code access to their system, or parts thereof, through WASI. Similarly, Wasmtime [6] is a standalone WebAssembly runtime that implements WASI and allows running WebAssembly code like any other native application.

3.4 Use Cases of WebAssembly in Node.js

Like native code, WebAssembly has the potential to be much faster than JavaScript (see Section 2.3). It is not subject to garbage collection, and has usually already been optimized by an optimizing compiler ahead-of-time. Unlike JavaScript, it is also an excellent compilation target for low-level languages. However, unlike native code, WebAssembly provides the portability of JavaScript, and similar security properties. Therefore, portable WebAssembly modules can provide good performance for CPU-

	JavaScript	WebAssembly	Native code
<i>Performance</i>	Medium	Good	Best
<i>Portability</i>	Good	Good	Very limited
<i>Security</i>	Good	Medium	Bad
<i>Sandboxing</i>	Possible	Yes	Requires virtualization
<i>Hardware features</i>	No	Limited	Yes
<i>As compilation target</i>	Bad	Good	Good

Table 3.1: Qualitative comparison of JavaScript, WebAssembly, and native code in Node.js

intensive tasks, such as numerical algorithms, media processing, data compression, image synthesis, and scientific computations.

Another reason to use WebAssembly is to avoid rewriting existing code for use with Node.js. Instead of converting the code to JavaScript or wrapping it in a native addon, low-level code can often simply be compiled to WebAssembly, and then used from Node.js directly (see Section 3.5.1).

Third, as described in Section 3.1, JavaScript has benefited from the ability to use the same code on different platforms, e.g., in client-side web applications and server-side Node.js applications, and this will likely also be true for WebAssembly.

Based on the previous sections and Chapter 2, Table 3.1 shows a qualitative comparison of JavaScript, WebAssembly, and native code in Node.js.

3.4.1 Side-channel Attacks Against Cryptographic Procedures

A particular use case for embedding C or other low-level code into JavaScript applications is cryptography. While Node.js provides a cryptography module, browsers only provide limited cryptographic features via the Web Cryptography API [17, 38]. Watt et al. argue that these available frameworks are insufficient for modern applications [70]. Much earlier, this fact had already led to JavaScript implementations of cryptographic routines, however, JavaScript turned out to be far from ideal for implementing such algorithms [17, 57], and can lead to insecure implementations [70]. As discussed above, WebAssembly is a better target for low-level code than JavaScript and provides more predictable performance, which is why Protzenko et al. suggest using WebAssembly as a new platform for cryptographic code [52]. However, as shown by Watt et al., the translation from WebAssembly code to instructions for the target architecture can introduce side-channel vulnerabilities, especially timing side-channel leaks due to optimization during compilation [70]. Since this compilation step cannot be controlled by the developer, but depends on the WebAssembly interpreter/compiler of the respective runtime, this risk is almost impossible to mitigate. The researchers proposed an extension of the WebAssembly specification that allows constant-time operations and reduces the risk of side-channel attacks [70]. However, at the time of writing, this extension is not readily available in WebAssembly compilers or runtimes.

3.5 Integration into JavaScript Applications

This section describes the integration of WebAssembly code into Node.js applications, as well as resulting issues, and relevant restrictions.

3.5.1 WebAssembly JavaScript Interface

The *WebAssembly JavaScript Interface* specification [67] was released along with the first version of the WebAssembly specification [66]. It defines how JavaScript applications can embed WebAssembly modules, and how they can interact with WebAssembly instances (see Section 2.2).

The following WebAssembly module imports a function `$multiply` from the namespace “env”. It declares and exports linear memory with an initial size of four pages (see Section 2.2), and a function `square`, which takes a 32-bit integer `x` and returns `$multiply(x, x)`.

```
(module
  (import "env" "multiply"
    (func $multiply (param i32) (param i32) (result i32)))
  (memory (export "memory") 4)
  (func (export "square") (param $x i32) (result i32)
    (call $multiply (local.get $x) (local.get $x))))
```

In the following, `wireBytes` refers to the binary encoding of this WebAssembly module, which has a size of 82 bytes:

```
const wireBytes = new Uint8Array(
  [0x00, 0x61, /* 78 more bytes */, 0x00, 0x0b]);
```

In Node.js, WebAssembly modules can be created from this binary representation in two ways. This step validates the module and compiles it such that it can subsequently be used. First, the constructor of the `WebAssembly.Module` class provides a synchronous way of creating a WebAssembly module, meaning that it only returns when compilation has finished, and blocks the calling thread until that happens.

```
const module = new WebAssembly.Module(wireBytes);
```

Alternatively, the `WebAssembly.compile` function asynchronously compiles the module in the background. It returns a `Promise` object immediately, which represents work that is being completed in the background, and which invokes its “then” callback when that work, i.e., the compilation, has finished [13]. The result of the compilation (i.e., the `module`) is passed to the “then” callback. Due to this asynchronous behavior, any code after the call to `compile` potentially executes before the callback.

```
WebAssembly.compile(wireBytes).then(function(module) {  
    // Use the module here.  
});
```

In either case, the returned `module` variable is now an instance of the JavaScript class `WebAssembly.Module`. As discussed in Section 2.2, a WebAssembly module describes structure and behavior, but to execute the code, a WebAssembly instance of the module is necessary. For example, the obtained `module` is unaware of any implementation of the imported `multiply` function, which would be required to call the `square` function. Imports can be passed as an object containing namespaces. In this case, we will only declare the “env” namespace, which contains a single function.

```
const importObject = {  
  env: {  
    multiply: (x, y) => x * y  
  }  
};
```

To create a WebAssembly instance from the obtained module, the WebAssembly JavaScript Interface again provides one synchronous and one asynchronous option. For brevity, only the synchronous API is presented here.

```
const instance = new WebAssembly.Instance(module, importObject);
```

The returned `instance` is now bound to the specific imports, and provides its own exports via `instance.exports`. The `square` function can now be called directly, and the WebAssembly JavaScript Interface converts between the JavaScript `number` type and the WebAssembly `i32` type automatically.

```
instance.exports.square(5) // returns 25
```

Note that the return value depends on the implementation of the JavaScript function `multiply` that was imported by the WebAssembly module. If, instead of the above implementation of `multiply`, the function `(x, y) => x + y` had been passed to the WebAssembly instance during its creation, the `square` function would return $5 + 5 = 10$ instead of $5 * 5 = 25$. The ability to change the behavior of WebAssembly code through imported functions without changing the WebAssembly code itself is an important portability and security aspect, see Chapter 2 and Section 3.3.

Not only functions can be exported, but also global variables, function tables, and linear memory objects. Indices into function tables are the WebAssembly equiva-

lent to function pointers in low-level languages, and can be used to pass “function pointers” to WebAssembly functions between JavaScript and WebAssembly. Linear memory objects represent the linear memory of a WebAssembly instance. In this example, the module exports its own linear memory, which is an instance of the `WebAssembly.Memory` class in JavaScript, and allows direct read/write memory access from JavaScript through its `buffer` property.

```
const exportedMemory = instance.exports.memory;
const size = exportedMemory.buffer.byteLength;
```

Since the WebAssembly module declared its linear memory object with an initial size of four pages, the JavaScript variable `size` now has the value $4 * 65536 = 262144$.

As we have seen in this section, the WebAssembly JavaScript Interface makes interaction between JavaScript and WebAssembly relatively simple. Note that, at the time of writing, Node.js does not implement the WebAssembly Web API [68], which was released along with the WebAssembly JavaScript Interface, and provides functions for streaming compilation. These allow JavaScript applications in web browsers to compile WebAssembly code while it is being received over a network stream, without requiring to fetch it entirely before beginning compilation, which reduces delays. However, Node.js applications usually load code from a local file system, which makes streaming compilation less advantageous than in a web browser, where code is loaded over the network. The primary reason for a missing implementation of the WebAssembly Web API in Node.js is the fact that Node.js does not implement required web technologies [68], which were designed for web browsers. Future versions of Node.js might implement these required technologies and the WebAssembly Web

API, which would improve compatibility between JavaScript code running in web browsers and JavaScript code within Node.js.

3.5.2 Data Access from WebAssembly

The limited access that WebAssembly has to its environment can be a severe restriction of interaction between WebAssembly and JavaScript. While it is a crucial security feature, it makes accessing JavaScript values from WebAssembly difficult. JavaScript can only pass fixed numbers of scalar values to WebAssembly, and vice versa. Large data structures and unstructured data can only be transferred through linear memory (see Section 3.5.1). Copying data from and to linear memory can be a significant performance burden and complicates function calls.

3.5.2.1 Example

We will demonstrate this problem using an example. Assume a WebAssembly module implements and exports the equivalent of the C standard library function `strstr`, which locates the first occurrence of a string `needle` within a string `haystack`, and returns a pointer to the occurrence [26]. There is no need to use WebAssembly for this, of course, and it would be trivial to implement the same behavior in JavaScript. However, this example will highlight some of the problems. From JavaScript, the function can be called as follows:

```
const result = instance.exports.strstr(haystack, needle);
```

However, `haystack` and `needle` cannot be JavaScript strings because only scalar values can be passed to and from WebAssembly. Additionally, even if a pointer to

the memory representing a string could be passed directly, the JavaScript memory model stores strings as sequences of UTF-16 code units, whereas the memory model of the C standard library does not [13].

In order for WebAssembly to be able to access both strings, they must reside in the linear memory of the WebAssembly instance. However, while JavaScript can access linear memory, it generally cannot know the linear memory layout chosen by the WebAssembly instance. For example, some (or most) parts of the linear memory might already be in use as part of the WebAssembly module’s own memory management, e.g., as part of the “unmanaged stack” or as part of dynamic memory allocations (see Section 2.2.2). While the WebAssembly module could reserve a statically allocated memory area within its own linear memory for such data exchange, this would enforce a static limit on the size of data that can be exchanged. Therefore, it is often necessary for WebAssembly to export its own memory management functions so the embedding application can use them. In this case, we will assume that the WebAssembly module exports functions equivalent to the C standard library functions `free`, `malloc`, and `strstr` [26], which can then be used in JavaScript:

```
const { free , malloc , strstr } = instance . exports ;
```

In our example, the embedding JavaScript code first needs to encode the JavaScript strings `haystack` and `needle` into byte sequences:

```
const encoder = new TextEncoder ();  
const hBytes = encoder . encode ( haystack );  
const nBytes = encoder . encode ( needle );
```

While JavaScript itself uses UTF-16 internally [13], the `TextEncoder` class encodes strings as UTF-8 sequences, which are compatible with the C standard library.¹ Since these byte sequences were newly allocated outside of WebAssembly linear memory, they are still inaccessible to WebAssembly. In order for WebAssembly code to be able to access them, they must be copied to linear memory, which requires allocating sufficiently large memory sections within linear memory first:

```
const hPtr = malloc(hBytes.byteLength + 1);
if (hPtr === 0) throw new OutOfMemoryError();
try {
    const nPtr = malloc(nBytes.byteLength + 1);
    if (nPtr === 0) throw new OutOfMemoryError();
    try {
        // Continue here.
    } finally {
        free(nPtr);
    }
} finally {
    free(hPtr);
}
```

As this simple case demonstrates, accessing low-level memory management within WebAssembly involves a substantial amount of error handling and manual cleanup. After having acquired both pointers, which are relative to the beginning of the instance's linear memory, JavaScript can safely copy the data to the locations indicated

¹This is only true for strings that do not contain the null character (`'\0'`). Unlike JavaScript strings, C strings cannot contain the null character [26].

by the pointers. Since the length of a C string is determined based on a terminating null character, the embedding JavaScript code must ensure that terminating null characters are added, which is why we intentionally allocated room for one additional byte per string in the previous code snippet. To be able to access and modify bytes within the linear memory, we need to use the `Uint8Array` adapter class:

```
const memoryView = new Uint8Array(linearMemory.buffer);
memoryView.set(hBytes, hPtr);
memoryView[hPtr + hBytes.byteLength] = 0;
memoryView.set(nBytes, nPtr);
memoryView[nPtr + nBytes.byteLength] = 0;
```

At this point, the C strings can be accessed from WebAssembly code, and the embedding JavaScript code can now simply pass the pointers to the C function:

```
const resultPtr = strstr(hPtr, nPtr);
```

Finally, the result can be converted to a JavaScript string again:

```
if (resultPtr === 0)
    return null;
const length = hBytes.byteLength - (resultPtr - hPtr);
const resultBytes = new Uint8Array(linearMemory.buffer,
                                   resultPtr, length);
const decoder = new TextDecoder();
return decoder.decode(resultBytes);
```

3.5.2.2 Complex Data Structures

In the previous example, we only considered strings, whose memory layout is simple compared to most other data structures in high-level programming languages. In fact, similar to different character encodings for strings, most complex data structures do not have a universal representation as byte sequences. For example, unlike in high-level programming languages, such as Java [46] and JavaScript, arrays in C do not contain information about their own dimensions [26].

Even if WebAssembly code was able to access memory belonging to its embedding application, it would not be able to use the data structures, especially not in a portable manner. This means that passing any non-trivial amount of data to WebAssembly is inherently difficult. Even simple byte sequences and strings, which only need to be copied to linear memory, require interaction with the internal memory management of the WebAssembly instance, e.g., by allocating and freeing memory dynamically. Passing more complex structures, e.g., C-style `struct` types [26], from JavaScript to WebAssembly (or vice-versa) requires precise knowledge of the memory layout of the structure within the WebAssembly instance's linear memory. Data types with dynamic structure, which are common in high-level languages, such as JavaScript [13], bring even more complexity with them.

An attempt to solve this problem is the *Interface Types Proposal* [8, 69], which adds types and instructions to WebAssembly to convert between data structures of the embedding application and data structures of the embedded WebAssembly module. However, while this proposal makes passing data from and to WebAssembly more convenient, it supports a limited set of complex types only, and adds significant

complexity to the WebAssembly specification. The proposal also still requires the WebAssembly module to implement dynamic memory management. At the time of writing, it is at stage “Phase 1 - Feature Proposal (CG)” [73], which means that it is still far away from being included in the WebAssembly specification.

An alternative is presented in Section 3.5.4, however, it is specific to JavaScript, and thus limits portability of WebAssembly code.

3.5.3 Memory Access from JavaScript

Many computationally intensive algorithms produce large results that are stored in memory. When compiled to WebAssembly, such results must be stored in linear memory. We will attempt to demonstrate an important restriction of linear memory access from JavaScript with an example. We will assume a WebAssembly module that was generated from C code, and which defines a function whose return value is a pointer to memory that was dynamically allocated using the `malloc` function.²

The following C function allocates 1024 KiB of memory and fills the allocated area with the given byte, assuming the allocation succeeded, otherwise, it returns the `NULL` pointer.

²While WebAssembly itself does not provide dynamic memory management, the C standard library does, and tools such as Emscripten [76] allow WebAssembly code to use the C standard library. Users may also implement their own memory management within the WebAssembly module, see Section 2.2.

```

unsigned char* fill_1mib(unsigned char b) {
    size_t size = 1024 * 1024;
    char* mem = malloc(size);
    for (size_t i = 0; mem != NULL && i < size; i++)
        mem[i] = b;
    return mem;
}

```

When invoked from JavaScript, the C function will return the pointer as a 32-bit integer. Since, from the perspective of WebAssembly, the address range of linear memory always begins at zero and is continuous, the JavaScript code can interpret the pointer as an offset into the linear memory of the WebAssembly instance.

As described in Section 3.5.1, JavaScript code can access the linear memory through the *WebAssembly JavaScript Interface*. For this purpose, JavaScript uses the built-in `ArrayBuffer` class, which represents allocated memory. This `ArrayBuffer`, combined with the returned pointer `ptr`, allows JavaScript to access the result directly:

```

const byte = 'w'.charCodeAt(0);
const ptr = instance.exports.fill_1mib(byte);
const mem = instance.exports.memory;

```

The calling JavaScript code could now obtain a copy of the memory area using `mem.buffer.slice(ptr, ptr + 1024 * 1024)`. However, if the result is large (in this example, 1024 KiB), copying it to a new memory area can become a significant performance problem, depending on how often the function needs to be invoked.

At first, it might seem that a simple solution exists: Since the WebAssembly code already allocated a part of its own linear memory for the result, it would be conve-

nient and efficient to keep referring to the relevant part of its linear memory instead of copying the data to a new location. This remains possible through the obtained linear memory object `mem`, however, this approach quickly leads to problems.

First, the `ArrayBuffer` representing the linear memory may be “detached”. This may happen, for example, when the size of the linear memory changes. Because the ECMAScript language specification requires the size of an `ArrayBuffer` to be unchangeable, a new `ArrayBuffer` object is created to represent the same linear memory, but with the new size. The old instance of the `ArrayBuffer` will be marked as “detached” and will not allow access to the linear memory anymore. Any function that refers to the WebAssembly instance’s linear memory would have to handle this case, and would need access to the WebAssembly instance to retrieve the new `ArrayBuffer`, which it could then use to access the result. This adds unrealistic complexity to any function that would otherwise use simple standard methods to access memory.

The second problem is related to memory management. The memory was dynamically allocated within WebAssembly, meaning that it cannot be reused for other purposes. Multiple calls to the function `fill_1mib` allocate 1 MiB of memory each, which can quickly lead to resource exhaustion. To avoid memory leaks, the memory should be deallocated by calling `free` from within WebAssembly when it is not needed anymore. This needs to be ensured by the calling JavaScript code, which usually relies on garbage collection for memory management. If `free` is called too late, or not at all, the application might end up using too much memory. If `free` is called before the result is not needed anymore, the application will use a “dangling pointer” on subsequent accesses, which can become a security risk.

Both of these problems might appear manageable in own code, but as soon as the allocated memory area must be accessed by a built-in or third-party library function, managing WebAssembly memory allocations from JavaScript becomes difficult.

This limitation is an unfortunate consequence of the stark differences between the WebAssembly and JavaScript memory models. As we have seen, not only does accessing memory owned by JavaScript from WebAssembly require copying, but the other direction usually requires copying, too. For performance reasons, data exchange between JavaScript and WebAssembly should therefore be minimized.

3.5.4 Reference Types

With the *Reference Types Proposal for WebAssembly* [54], which is implemented in recent Node.js versions, it is possible to pass JavaScript values to WebAssembly functions in the form of opaque references. While WebAssembly code itself is unable to perform any operations on these values, whose type within WebAssembly is `externref`, it can pass such values to imported JavaScript functions, or return them to the calling JavaScript function. For example, a JavaScript application could pass a JavaScript object to a WebAssembly function, and the WebAssembly function could use imported JavaScript functions to extract information from the object.

WebAssembly code should be portable, and reference types do not violate this design goal. From the perspective of WebAssembly, it is irrelevant whether a value of type `externref` is a JavaScript, Java, Python, or C# object, and whether these values are passed to JavaScript, Java, Python, or C# functions. It is up to the embedding application or runtime to assign semantics to references.

3.5.4.1 Using the Reflect Interface

However, this does not prevent WebAssembly modules from being designed for a specific runtime. For instance, for interaction between WebAssembly and JavaScript runtimes, such as Node.js, the general-purpose functions provided by JavaScript's `Reflect` object [13] are of particular interest. When imported into a WebAssembly instance, these functions allow WebAssembly code to modify and retrieve information from JavaScript values. For example, to check for the existence of, to retrieve, and to set array elements, the `Reflect.has`, `Reflect.get`, and `Reflect.set` functions can be imported, respectively.

```
(import "Reflect" "has"
  (func $array_has (param externref) (param i32)
    (result i32)))
(import "Reflect" "get"
  (func $array_get (param externref) (param i32)
    (result externref)))
(import "Reflect" "set"
  (func $array_set (param externref) (param i32)
    (param externref)))
```

The following WebAssembly function demonstrates how to use these imported functions to copy all values from one JavaScript array to another.

```

(func $js_array_copy (param $source externref)
                    (param $dest externref)
  (local $i i32)
  (loop $while
    (block $end
      (br_if $end (i32.eqz (call $array_has (local.get $source)
                                          (local.get $i))))
      (call $array_set (local.get $dest) (local.get $i)
                (call $array_get (local.get $source) (local.get $i)))
      (local.set $i (i32.add (local.get $i) (i32.const 1)))
      (br $while))))

```

Note that the WebAssembly code in this example does not explicitly mention any JavaScript types, in particular not the `Array` class. From the perspective of WebAssembly, `source`, `dest`, and any values contained in either array are opaque references. This means that, even if the values in the source array refer to JavaScript numbers whose values could be represented within WebAssembly, there is no way for WebAssembly code to perform arithmetic on them.

To facilitate operations on JavaScript values, it is useful to import additional helper functions. For example, the identity function is surprisingly useful when imported:

```

const helper = {
  identity(x) {
    return x;
  }
};

```

Since JavaScript is agnostic to the type of `x`, WebAssembly can import the same function multiple times with different function types. For example, the following two import declarations both use the same `identity` function and allow “unboxing” numeric JavaScript values. Effectively, these functions convert from opaque references to the WebAssembly types `i32` and `f64`, respectively.

```
(import "helper" "identity"  
  (func $ref_to_i32 (param externref) (result i32)))  
(import "helper" "identity"  
  (func $ref_to_f64 (param externref) (result f64)))
```

When called with a value `x` of type `externref`, the WebAssembly JavaScript Interface (see Section 3.5.1) will pass the reference to the `identity` function, and convert the return value, which is the same as the argument `x`, to the WebAssembly type that was specified in the import declaration.

For example, the following WebAssembly function uses `$ref_to_f64` to compute the sum of the elements of a JavaScript array as a 64-bit floating point number.

```

(func $sum (param $array externref) (result f64)
  (local $i i32)
  (local $sum f64)
  (loop $while
    (block $end
      (br_if $end (i32.eqz (call $array_has (local.get $array)
                                                (local.get $i))))
      (local.set $sum
        (f64.add (local.get $sum)
          (call $ref_to_f64
            (call $array_get (local.get $array)
                          (local.get $i))))))
      (local.set $i (i32.add (local.get $i) (i32.const 1)))
      (br $while)))
  (return (local.get $sum)))

```

Similarly, helper functions can be added to create, modify, and retrieve JavaScript values from WebAssembly beyond the capabilities of the `Reflect` interface.

As we have seen, reference types simplify interaction between JavaScript and WebAssembly tremendously without sacrificing the type safety of WebAssembly. However, to ensure this type safety, it is impossible to store references, that is, values of type `externref`, in linear memory, which can be a significant restriction.

3.5.5 Blocking versus Non-blocking Behavior

JavaScript applications have traditionally been single-threaded, meaning that JavaScript code could not perform multiple computations at the same time. A long-running computation could make the entire application, e.g., on the client-side, a web page, or, in the case of Node.js, an entire web server, unresponsive. All other operations would have to wait until the current computation had finished. Node.js attempted to mitigate this problem through the `cluster` module, which allows multiple processes to communicate, and performs load-balancing by assigning different requests to different application processes. However, each process could still only run a single application thread, and a long-running computation could still make an entire process unresponsive.

Node.js was originally not designed for computationally intensive tasks. Its software library provides non-blocking implementations of many functions that could otherwise block the thread for a considerable amount of time. For example, all input/output activity can be achieved through asynchronous functions, which notify the application thread when their result is ready, without blocking the application thread [60]. Internally, asynchronous functions use a thread pool to complete work without interfering with the application. However, these internal threads cannot run JavaScript (or WebAssembly) code, and changing that would not solve the problem, since any long-running computation would then prevent input/output activity.

In the following example, the `readFile` function schedules reading a file in a background thread, and will later invoke the given function (“callback”) with an error or the contents of the file, depending on whether the operation was successful. How-

ever, until that result is available, the application can perform other tasks. In this case, “Hello world” would be printed before the callback is invoked.³

```
readFile('/etc/passwd', function(error, data) {  
  if (error) throw error;  
  process.stdout.write(data);  
});  
console.log('Hello world');
```

However, asynchronous callbacks are only invoked once the application thread is idle. For example, if the previous example followed the call to `readFile` with an infinite loop instead of printing “Hello world”, the thread would never be idle, and the callback would never be invoked.

Interested readers are referred to Loring et al. for a formalization of the asynchronous programming model employed by Node.js and JavaScript in general [33].

3.5.5.1 Communication Difficulties

WebAssembly code has no built-in concept of asynchronicity: A WebAssembly function will block the calling application thread for its entire duration. Unlike Node.js, WebAssembly was designed for computationally intensive tasks, which makes long-running functions almost unavoidable. In other words, JavaScript code implements non-blocking behavior, whereas WebAssembly implements blocking behavior.

Modern web browsers and recent versions of Node.js provide *Web workers* and *Worker threads*, respectively [45]. The motivation behind these technologies is the

³This callback-based concept of asynchronicity was also described for the `WebAssembly.compile` function in Section 3.5.1.

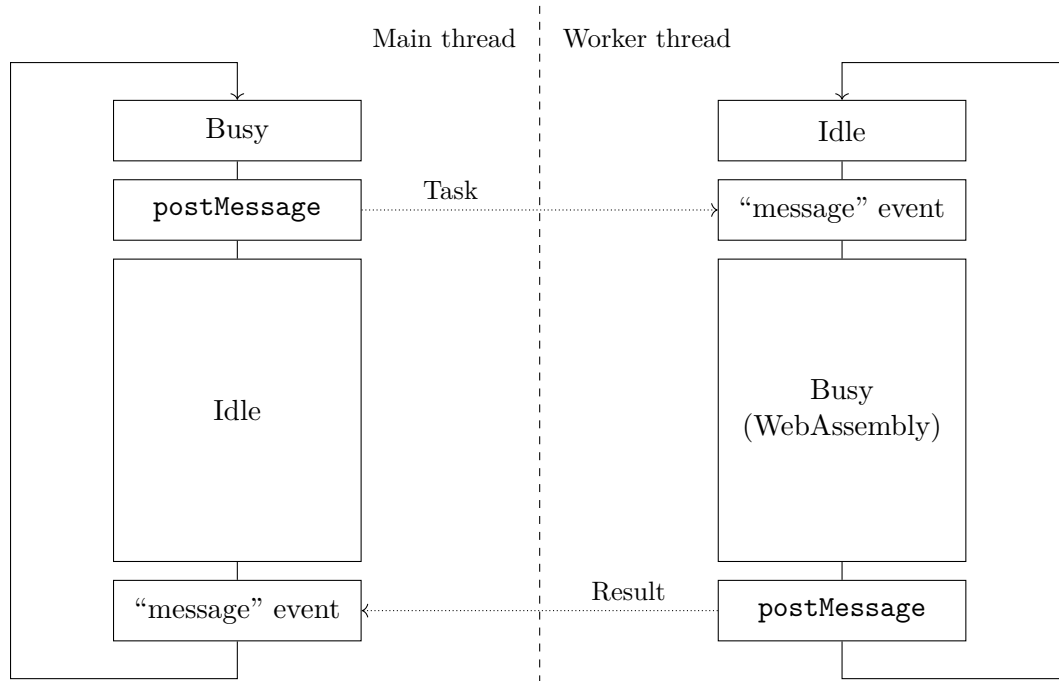


Figure 3.1: Basic communication with WebAssembly in a worker thread

ability to run computationally intensive tasks in additional application threads, which can communicate with the main application thread, and which, therefore, will not prevent the main thread from completing other work [50].

Both in web browsers and in Node.js, the primary way of communication between application threads is the JavaScript `postMessage` function, which sends a message to another thread. Similar to the callback in the previous JavaScript example, messages are received asynchronously. If an application thread is busy with synchronous work (“blocked”), it will not receive messages until after the synchronous work has been completed, and the thread is idle.

This works well if the WebAssembly code running in the worker thread does not need to interact with the main thread, or if it only posts results to the main thread.

By moving the computationally intensive parts to a worker thread, the main thread can remain idle, and thus responsive (see Figure 3.1). If the main thread requires multiple tasks to be completed by a WebAssembly module, it can send multiple messages to the same worker thread, which will process them sequentially: As soon as the worker thread becomes idle, it will receive one of the queued messages, complete the task, report the result, become idle again, and receive another message, until all messages have been processed. Alternatively, the main thread can create additional worker threads, which can then each receive messages and complete tasks. While all threads use the same WebAssembly module, each thread uses a separate WebAssembly instance of the module.

However, this communication model fails when the WebAssembly code running in the worker thread requires information from the main thread during its execution, e.g., user interaction. While WebAssembly code can import a JavaScript function to invoke `postMessage` and notify the main thread, it cannot asynchronously receive messages, because it runs synchronously and is thus never fully idle from the perspective of JavaScript. During the execution of WebAssembly code, it is, therefore, impossible for the thread to receive messages asynchronously. Due to the single-threaded event model of JavaScript, asynchronous communication is the only fully supported communication model.

3.5.5.2 Implementation of Synchronous Communication

However, Node.js and some web browsers provide basic means of sharing memory between threads using the `SharedArrayBuffer` class. It represents allocated memory that can be accessed by multiple threads in a thread-safe manner using atomic oper-

ations. At the time of writing, this is the only possibility to synchronously exchange data between threads [13, 45].

The software library *synchronous-channel* demonstrates this possibility [39]. It was developed during our research, and published as an open-source project. The library provides a `SynchronousChannel` class, which represents a unidirectional message queue. A pair of channels can be used for bidirectional communication.

Internally, each `SynchronousChannel` uses a fixed-size `SharedArrayBuffer`, on top of which it implements a ring buffer. The message queue is then implemented on top of the ring buffer. Since `SharedArrayBuffer` instances cannot be resized, after creation of a `SynchronousChannel`, the maximum number of messages and the maximum size of each message in the queue cannot be changed.

The library implements writing to and reading from a `SynchronousChannel` both with non-blocking and blocking behavior. In non-blocking mode, a call to `read` returns immediately, even if no messages are available, and a call to `write` returns immediately, even if the message cannot be written because the queue is full. In these cases, the return values of the functions indicate these error conditions.

If a timeout is passed to `read` or `write`, the functions switch to their blocking modes. For `read`, this means that the function returns as soon as a message can be read from the queue, or when the given timeout elapses. Similarly, `write` blocks the calling thread until there is room in the message queue for the message to be written, or until the timeout elapses. An infinite timeout may be specified, in which case both functions may block indefinitely.

For example, a WebAssembly module might import a JavaScript function that requires information from the main application thread, but cannot use asynchronous

<i>Sending thread mode</i>	<i>Receiving thread mode</i>	<i>Mechanism</i>
Asynchronous	Asynchronous	<code>postMessage</code>
Asynchronous	Synchronous	<code>SynchronousChannel</code>
Synchronous	Asynchronous	<code>postMessage</code>
Synchronous	Synchronous	<code>SynchronousChannel</code>

Table 3.2: Suggested inter-thread communication model between synchronous (WebAssembly or JavaScript) and asynchronous (JavaScript) tasks

behavior and, therefore, not receive messages asynchronously. However, it can still send a request for information to the main thread using `postMessage`, since the function only requires the receiving thread to act asynchronously (see Table 3.2). After sending the request, the imported JavaScript function synchronously waits for a message on a `SynchronousChannel`. While this does block the thread, it requires virtually no resources, since the “wait” operation is implemented using low-level synchronization mechanisms, which cause the thread to be entirely inactive until resumed by an incoming message. Once the main thread receives the request asynchronously via a “message” event, it can write its response to the `SynchronousChannel`. This action resumes the waiting JavaScript function in the other thread, which can now return the obtained information to WebAssembly.

While the `SynchronousChannel` class itself implements thread-safe behavior, classic thread synchronization issues, such as deadlocks, can occur in the application. For example, if the receiving thread stops reading messages from a channel, the message queue will eventually be filled with pending messages, and the sending thread will be unable to write additional messages. Such issues, however, are not specific to this communication model, or this software library. For example, the `postMessage`

function for asynchronous communication will similarly fail to deliver messages if the receiving thread is unable to receive messages, e.g., because it is never idle.

In conclusion, a `SynchronousChannel` allows unidirectional or bidirectional synchronous communication between threads, and implements both blocking and non-blocking behavior. The main thread of an application could use such a channel to send messages to WebAssembly code running in another thread, e.g., to facilitate user interaction with the secondary thread.

3.6 Porting N-API to WebAssembly

Node.js provides *N-API* as a framework for native addon developers to simplify interaction between C/C++ and JavaScript, V8, and Node.js internals [43]. For example, it defines types and functions that allow C/C++ code to work with JavaScript values and objects, and native addons can use N-API to define functions that are implemented in native code, but can be called from JavaScript.

We already compared WebAssembly to native addons in Section 3.2. Here, we would like to briefly consider how to port existing C/C++ code that uses N-API to WebAssembly. In this context, we will have to distinguish between different kinds of code. We will refer to the WebAssembly instance that uses N-API and replaces the native addon as the *WebAssembly addon*. With *native code*, we mean other native addons and native code included in Node.js and V8. With *managed code*, we mean non-native code that is executed by V8, i.e., JavaScript and WebAssembly code.

From the very beginning, it becomes apparent that this attempt goes against the goals of WebAssembly. WebAssembly code is supposed to be portable and should

not make strong assumptions about its environment, i.e., it should not only work when embedded into a Node.js application. As we will see below, the design of N-API also does not align well with the design of WebAssembly. Since there is interest in this topic, it is presented here, but only briefly.

The need for a library, such as N-API, is much smaller in WebAssembly than in native code. Unlike native functions, WebAssembly functions can be called from JavaScript directly (see Section 3.5.1). Pointers are represented as numbers in JavaScript, and can be used to access WebAssembly linear memory. Many C libraries can simply be compiled to WebAssembly and used directly from JavaScript, without any “glue code” written in C or other low-level languages. This removes the need for N-API in many cases, and importing custom JavaScript functions is often a suitable replacement. Where it does make sense for WebAssembly functions to actively interact with JavaScript instead of only being called from JavaScript, reference types are likely a better choice for WebAssembly modules (see Section 3.5.4) than attempting to use N-API.

N-API is a part of Node.js and consists of a set of native functions that are meant to be used by native C/C++ addons. Since WebAssembly cannot call native functions directly, but only imported JavaScript (or WebAssembly) functions, a WebAssembly addon cannot access native N-API functions directly. As we will see in the following section, even if it was possible to access the native N-API implementation from WebAssembly, V8’s execution model would prevent N-API function calls from working as expected. It becomes apparent that a new N-API implementation is required for WebAssembly addons.

3.6.1 Exception Handling

Managed code, with the exception of JavaScript code inside `finally` statements,⁴ cannot be executed while an exception is pending. An exception is pending until it is caught, or until it terminates execution. However, native code is allowed to run even if an exception is pending. Often, native addons themselves throw an exception, which is pending until control is handed back to V8 by returning from the native function. Before continuing the execution of managed code, V8 detects the pending exception and aborts the normal control flow, until an applicable exception handler (e.g., a JavaScript `catch` statement) has been found.

WebAssembly itself does not have the ability to handle exceptions. Therefore, any pending exception would abort WebAssembly execution upon returning from an N-API call. While this might often match the desired behavior, it is incorrect: for example, a native addon might intend to clean up resources after scheduling an exception, but before returning control to V8. However, if the same code was compiled to WebAssembly, the pending exception would abort execution of the WebAssembly addon, and would, therefore, also prevent resources from being cleaned up.

This problem is avoidable with a new N-API implementation. Whenever interaction with managed code could result in an exception, or when intentionally throwing an exception, the custom N-API implementation needs to catch the exception. It also needs to track the boundaries between the WebAssembly addon and other managed code. When switching from the WebAssembly addon to other managed code, the

⁴Code enclosed in a JavaScript `finally` statement is executed regardless of whether an exception was thrown and whether said exception is still pending or was caught [13]. Similar concepts exist in other high-level programming languages, such as Java [46], C# [27], and Python.

implementation needs to check whether it caught an exception since the last switch from other managed code to the WebAssembly addon. In this case, the exception is thrown again.

The primary difficulty is the tracking of boundaries between other managed code and the WebAssembly addon, especially since the WebAssembly instance might import and call managed code functions that do not belong to N-API. In such cases, it is often unclear whether a pending exception should be thrown or not.

3.6.2 Pointers to JavaScript Values

N-API represents references to JavaScript values as the opaque `napi_value` type, which is defined to be a pointer. It is not a problem that this pointer does not point to the linear memory of a WebAssembly instance, since it is never dereferenced from WebAssembly, but only passed to N-API functions.

On 64-bit architectures, these pointers will have a size of 64 bits. However, the WebAssembly JavaScript Interface (see Section 3.5.1) does not allow passing 64-bit integers from or to WebAssembly. This stems from a limitation of the data types in JavaScript. The JavaScript `number` type can hold 32-bit integers or 64-bit floating point numbers. However, not every 64-bit integer can be represented as a 64-bit floating point number, therefore, it is unsafe to use the JavaScript `number` type to exchange 64-bit integers.

Recent versions of Node.js allow passing 64-bit integers using the `BigInt` type, which can represent arbitrarily large integers. The Node.js version used in our experiments includes this as an “experimental feature.”

However, the size of the pointer is not the only issue. While an `napi_value` points to a JavaScript value, it does not prevent it from being garbage collected. In native addons, this is not a problem, because objects will not be garbage collected until the native addon returns control to V8. In a WebAssembly addon, this guarantee does not exist. The value may be garbage collected at any point, in which case the `napi_value` becomes invalid, i.e., a “dangling” pointer.

Again, an additional layer of abstraction can provide a workaround, and manage the lifetime of JavaScript values. Instead of using actual memory pointers, a custom N-API implementation can maintain an array of JavaScript values for each invocation of a function of the WebAssembly addon. Instead of being a pointer, an `napi_value` is now an index into the array. Since pointers of type `napi_value` only need to be valid during the execution of the addon function that obtained them, their implementation should not prevent JavaScript values from being garbage collected beyond that duration. Therefore, when all WebAssembly addon functions have returned control to JavaScript, meaning that no WebAssembly code that is using N-API is on the call stack anymore, the contents of the array can be deleted. This allows all contained JavaScript values to be garbage collected, unless they are referenced otherwise, e.g., because the WebAssembly addon created persistent references to them. Persistent references can be implemented analogously to the described implementation of the `napi_value` type.

One downside of this approach is that garbage collection can only occur once all WebAssembly addon functions have returned, even though one might be particularly long-running, and might, therefore, cause unnecessarily high memory usage. To solve this problem, the N-API implementation can maintain a stack of such arrays, each

corresponding to one WebAssembly add-on function that is currently on the call stack. In this case, a part of the index must refer to the stack entry, and the remainder to the index of the value within the array associated with that stack entry. For example, when using 32-bit “pointers”, the upper 8 bits can refer to the index of the stack frame, starting with zero for the lower-most stack frame, and the remaining 24 bits of the index can refer to the offset of the value in the array of JavaScript values of the stack frame. Upon receiving a pointer of type `napi_value`, the N-API implementation can then use it to locate the array containing the desired JavaScript value, and the value within the array.

Note that the previously discussed *reference types* (see Section 3.5.4) cannot be used as a replacement for the `napi_value` pointer type. Even though they represent references to JavaScript values and ensure proper object lifetimes, they cannot be stored in linear memory.

3.6.3 Access to Memory Allocated by Node.js

One of the most important restrictions stems from WebAssembly’s inability to access memory outside of its own linear memory. N-API, on the other hand, makes frequent use of pointers that point to data allocated by Node.js, and which, therefore, are not part of linear memory.

For example, the `napi_get_arraybuffer_info` function allows add-ons to retrieve a pointer to the data stored in an `ArrayBuffer`. However, a WebAssembly add-on would not be able to access memory at the returned pointer, since it is outside of linear memory.

This problem is virtually impossible to solve. The data could be copied to linear memory, but more issues arise if the addon attempts to modify the data, in which case it would have to be copied back. Apart from the enormous performance overhead, this requires N-API to perform complex memory management within linear memory.

3.6.4 Asynchronous Tasks

N-API allows scheduling work to be executed in a background thread, for example, to complete slow input/output operations. Node.js itself uses this concept to execute long-running tasks without blocking the calling application thread (see Section 3.5.5). However, V8 does not allow managed code to run outside of application threads, for example, because there is no obviously correct way to handle exceptions occurring in managed code outside of application threads.

A proposed addition to WebAssembly would allow multiple application threads, but not non-application background threads, to execute WebAssembly code [73]. If implemented, this proposal might allow simulating the behavior of asynchronous work provided by N-API, however, the implementation is unclear.

3.7 Summary

In this chapter, we have explored aspects of the use of WebAssembly code in Node.js applications. As we have seen, WebAssembly can provide a good platform for integrating code written in other languages and performance-critical code, without being limited to Node.js applications. Unlike native addons, WebAssembly modules do not require *glue code* to interact with JavaScript, which makes the use of

respective software libraries, such as N-API [43], mostly obsolete. However, we have also seen that data exchange between JavaScript and WebAssembly is limited by various factors, some of which are necessary to guarantee the security properties of WebAssembly. Other restrictions and difficulties are results of different memory and execution models across programming languages and hardware abstractions.

As described in Section 3.5.1, WebAssembly modules are loaded dynamically and at runtime in Node.js applications. The loaded WebAssembly code must either be interpreted or compiled, and the following chapter explores these processes in depth.

Chapter 4

Compilation at Runtime

As explained in Section 2.2, WebAssembly code needs to be interpreted or compiled at runtime because it does not target the actual physical machine, but a conceptual stack machine. This chapter discusses aspects of compilation and interpretation at runtime within Node.js and V8.

The compiler infrastructure within V8 has changed significantly in the last few years. Even for the relatively new WebAssembly language, V8 implements a complex combination of compilation procedures. The basic components are a WebAssembly interpreter, the baseline compiler *Liftoff* [18], and the optimizing compiler *TurboFan*, which V8 also uses to compile JavaScript [62, 59].

Since JavaScript code itself does not contain static type information, it is difficult to compile JavaScript code directly. For example, the following JavaScript function gives no information about the types of `a` and `b`:

```
function add(a, b) {  
    return a + b;  
}
```

To compile this function to machine code, type information about `a` and `b` is necessary. For example, the plus operator might be used to add two integers, or two floating point numbers, or to concatenate two strings, and each case would require different instructions to be generated. Due to this difficulty, V8 begins JavaScript execution using the *Ignition* interpreter, and only when the interpreter has identified “hot” (frequently executed) code sections, the TurboFan compiler is used to compile and optimize these JavaScript functions using type information gathered by Ignition. Since compilation happens during execution, this is referred to as *just-in-time compilation* (JIT).

WebAssembly, on the other hand, is not a high-level language, and not dynamically typed, and it is, therefore, not necessary to collect dynamic type information before compiling WebAssembly code [18]. For example, while the JavaScript function above uses the plus operator on two variables of unknown type, WebAssembly does not have such an operator, but specific instructions for different types (e.g., `i32.add` to add 32-bit integers, `f64.add` to add 64-bit floating point numbers, etc.), and static validation of WebAssembly code ensures that the operands always match the expected input types of these instructions (see Section 2.2). This lack of type ambiguity allows compiling WebAssembly code *ahead-of-time* (AOT), that is, before execution.

The TurboFan compiler is an optimizing compiler. It compiles and optimizes WebAssembly through a complicated pipeline that first decodes WebAssembly function

bodies and then constructs graph representations. These graph representations are in *Static Single Assignment form (SSA)* and use the “Sea of Nodes” concept introduced by Click in his dissertation [9]. TurboFan then applies optimizations to the SSA, before selecting appropriate instructions for the target architecture, allocating CPU registers, and finally generating code.

The Liftoff compiler, on the other hand, was designed to be fast at the cost of generating less optimized code. Even though it is newer than the TurboFan compiler, it is not meant as a replacement, but as the initial compilation stage to quickly produce a usable module. Like TurboFan, Liftoff begins by decoding WebAssembly function bodies, but then immediately begins code generation in a single pass, instruction by instruction, without constructing an SSA representation or optimizing the code [18]. The WebAssembly interpreter is not used by V8 by default, but can be enabled to debug WebAssembly code. We expect interpretation to be much slower than compiled code, and will test this experimentally in Section 4.2.

4.1 Compilation in the WebAssembly JavaScript Interface

As described in Section 3.5.1, from an application developer’s perspective, JavaScript applications can compile WebAssembly modules in two ways. The first is to call the constructor of the `WebAssembly.Module` class, which will synchronously compile the code, meaning that it will block the calling thread for the duration of the compilation. The second option is the asynchronous function `WebAssembly.compile`.

By default, `WebAssembly.Module` uses Liftoff to compile the code, which is the faster compiler, and thus causes the smallest delay in the calling thread. V8 compiles the same module again, in a set of background threads, using the optimizing TurboFan compiler. When the optimized compilation result for a WebAssembly function is ready, the next invocation of the function uses the code produced by the TurboFan compiler instead of the output of Liftoff. (However, existing function invocations are not updated, meaning that, if a function is currently part of the active call stack, it will continue to use the code produced by Liftoff.) This process is called “tiering-up”, and is a tradeoff between startup time and code generation quality [18].

The asynchronous `WebAssembly.compile` function, on the other hand, does not block the calling thread, and, therefore, does not prioritize fast compilation results. Its default behavior is to use the optimizing TurboFan compiler, skipping the baseline compilation step. We expect this to cause the compilation to generally take longer than synchronous compilation using Liftoff would, and will test this experimentally in Section 5.2. Unlike the synchronous tiering-up behavior, this produces the optimized result directly.

Note that, while this is an accurate description of the behavior that exists in the Node.js and V8 versions used in our experiments, it is not mandated by the WebAssembly specification [66], the WebAssembly JavaScript Interface specification [67], or the WebAssembly Web API specification [68], and might, therefore, change in future versions of Node.js or V8. Additionally, most behavioral aspects of WebAssembly compilation can be changed through process options that are passed to V8, and we will, at times, use this feature to test individual compilers and their performance separately.

For any two algorithms that attempt to solve the same problem, it is natural to compare two of their properties: Effectiveness and efficiency. Effectiveness is the degree to which an algorithm produces desirable results, whereas efficiency is the degree to which an algorithm uses minimal resources. As described in Section 2.7, existing research has not sufficiently distinguished between different compilers and compilation strategies in V8 and Node.js. In the following sections, we will consider the effectiveness of the compilers Liftoff and TurboFan, that is, their ability to produce fast instruction sequences from WebAssembly code. In practice, this effectiveness directly influences the execution time of the compiled code. The efficiency of the compilers will be explored in Section 5.2, which, in practice, translates to the resource usage of compilation. Based on the previously described design goals of both compilers, we expect Liftoff to be more efficient, but less effective, than TurboFan.

4.2 Performance of Generated Code

For the reasons described above, this section compares the effectiveness of TurboFan and Liftoff based on the performance of the generated code.

4.2.1 Experimental Methodology

In order to compare the performance of code generated by Liftoff to the performance of code generated by TurboFan, we compiled the *PolyBench/C 4.2* benchmarks [51] to WebAssembly with compiler optimization and Link Time Optimization (LTO) enabled. These benchmarks are scientific computing kernels and were already used by Haas et al. [15] and Jangda et al. [28] to compare the performance of WebAssembly

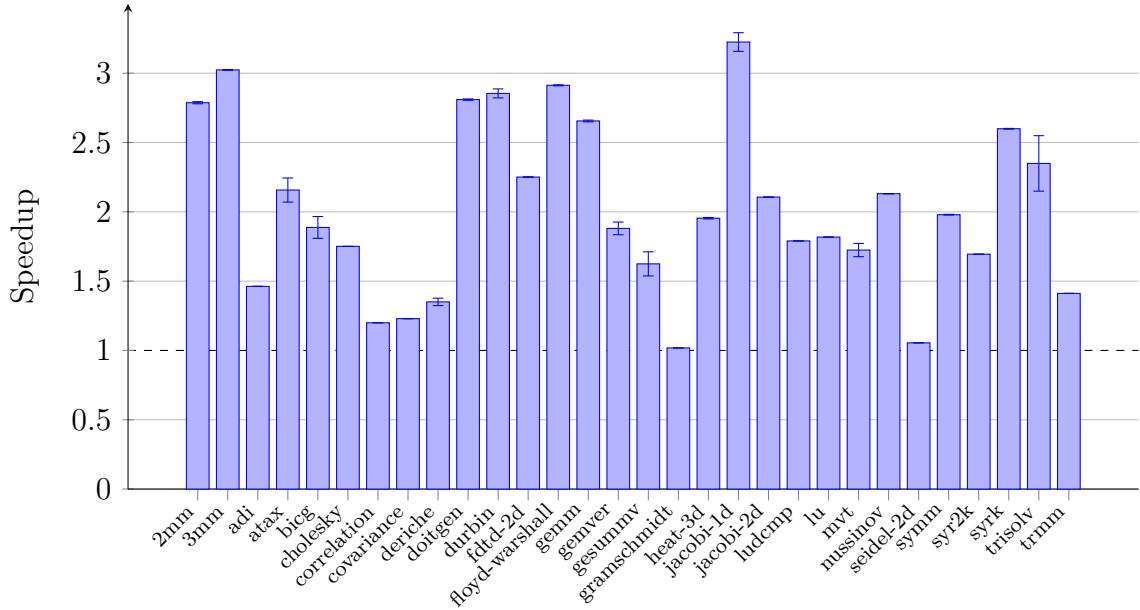
Behavior	V8/Node.js Process Options
TurboFan only	<code>--no-liftoff</code>
Liftoff only	<code>--liftoff --no-wasm-tier-up</code>
Interpreter only	<code>--wasm-interpret-all</code>

Table 4.1: Required process options for desired behaviors of synchronous WebAssembly module creation in V8

to the performance of native code execution. Instead, we use the benchmarks to compare the performance of code generated by Liftoff to the performance of code generated by TurboFan.

We conducted all experiments on Ubuntu 19.04 running on an Intel® Core™ i7-8700 processor (base frequency 3.20GHz, turbo frequency 4.60GHz, 6 cores, 12 threads) with 32GB of memory (2666 MHz). We used Node.js v14.2.0, the most recent Node.js version at the time of writing, which was released in May 2020, and which is based on V8 version 8.1.307.31-node.33.

For this experiment, we used the synchronous WebAssembly compilation interface (see Section 3.5.1) and enabled only one WebAssembly compiler (or the WebAssembly interpreter) at a time. The relevant process options for Node.js 14.2.0 are shown in Table 4.1 and are passed to the process that compiles and executes the respective benchmark. We compiled and ran each of the 30 PolyBench/C benchmarks one hundred times with only Liftoff enabled, and another one hundred times with only TurboFan enabled. We measured the time it took for the benchmarks to complete, which does not include their respective compilation times. In addition, we also ran all benchmarks using V8’s WebAssembly interpreter, which does not compile the WebAssembly modules.



PolyBench/C 4.2 benchmark

Figure 4.1: Speedup of code generated by TurboFan with respect to Liftoff

4.2.2 Results

Figure 4.1 shows the average speedup of the code generated by TurboFan with respect to the code generated by Liftoff for each benchmark, with error bars indicating one standard deviation in each direction. All benchmarks were faster when compiled with TurboFan, the average speedup across all benchmarks is 2.0, and the maximum speedup is 3.2. The geometric mean of the speedup across all benchmarks is 1.9.

Figure 4.2 shows the measured speedups of compiled code with respect to interpretation. On average, the PolyBench/C benchmarks were 247 times slower when interpreted than when compiled using TurboFan, and 115 times slower than when compiled using Liftoff. Sixteen of the 30 benchmarks were at least 200 times faster when compiled to TurboFan than when interpreted. We can infer that, while in-

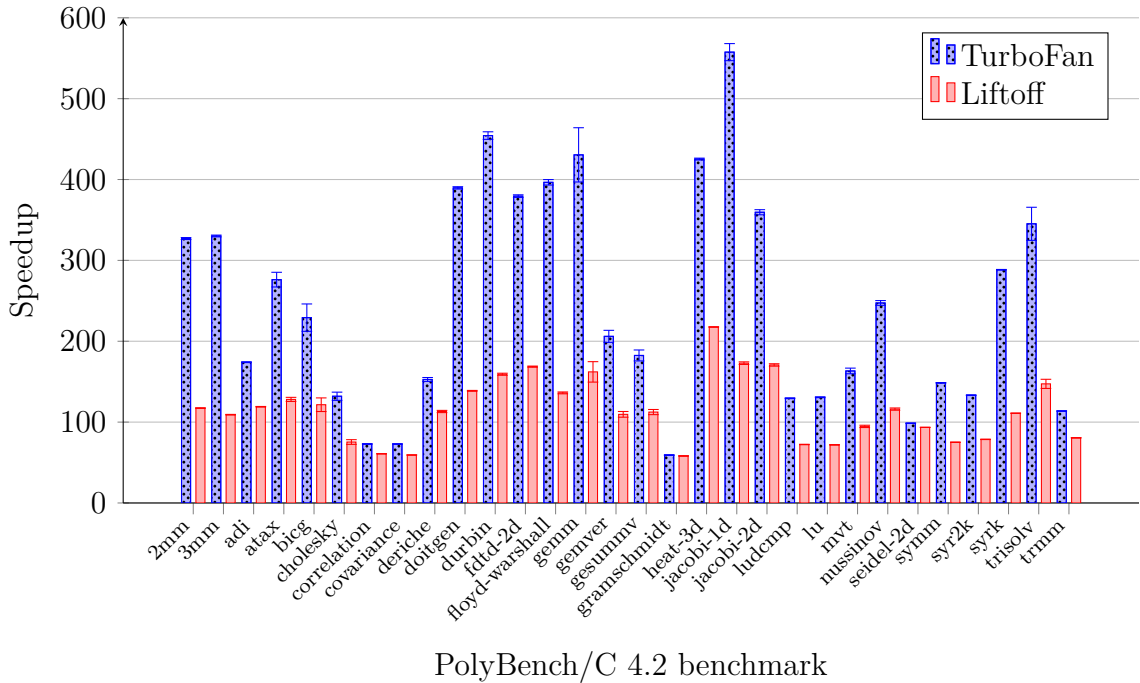


Figure 4.2: Speedup of compiled code with respect to interpretation

terpretation allows running WebAssembly code without prior compilation, its code execution is too slow for use in real applications.

We conclude that the optimized code generated by TurboFan is indeed significantly faster than code generated by the baseline compiler Liftoff, and that the code produced by both compilers is much faster than V8’s WebAssembly interpreter. However, PolyBench/C is not a good basis for testing the effectiveness of the compilers, that is, the performance of the compilers themselves, since the benchmarks are small and the WebAssembly modules are structurally very similar. Therefore, Section 5.2 uses a different set of WebAssembly modules to measure the efficiency of Liftoff and TurboFan in terms of time, memory, and CPU usage.

4.3 Quality of Generated Code

In the previous section, we observed performance differences between code generated by Liftoff and code generated by TurboFan. It is out of scope for this thesis to perform an in-depth analysis of the quality of code generated by WebAssembly compilation within V8. However, Jangda et al. found that one of the main reasons behind WebAssembly being slower than native code is the larger number of generated instructions, and especially branch instructions [28]. Therefore, we intend to compare the number of instructions and branch instructions for each of the benchmarks used in the previous section. The hypothesis is that code generated by Liftoff consists of more instructions than the code generated by TurboFan, and contains more branch instructions.

4.3.1 Example

Consider this text representation of a WebAssembly module that contains a single function “add2”, which takes two 32-bit integers and returns their sum.

```
(module
  (func $add2 (export "add2")
    (param $a i32) (param $b i32) (result i32)
    (return (i32.add (local.get $a) (local.get $b))))))
```

The TurboFan compiler produces the following instruction sequence for the function:

```
0    push rbp
1    movq rbp, rsp
4    push 0xa
6    push rsi
7    addl rax, rdx
9    movq rsp, rbp
c    pop rbp
d    retl
e    nop
f    nop
```

The function begins with a usual stack setup (offset 0 up to and including 0x6), and ends with a normal return sequence (beginning at 0x9), followed by `nop` instructions for alignment. The function arguments are passed in the `rax` and `rdx` registers. Conveniently, the caller expects the return value to be passed back in the `rax` register, which allows the compiled code to compute the result in a single `addl` instruction at offset 0x7.

The Liftoff compiler, on the other hand, produces this significantly longer sequence:

```
0    push rbp
1    movq rbp, rsp
4    push 0xa
6    subq rsp, 0x10
d    movq [rbp-0x10], rsi
11   movq rcx, [rbp-0x10]
```

```

15    movq rcx, [rcx+0x2f]
19    cmpq rsp, [rcx]
1c    jna 0x2c
22    leal rcx, [rax+rdx*1]
25    movl rax, rcx
27    movq rsp, rbp
2a    pop rbp
2b    retl
2c    push rax
2d    push rdx
2e    call WasmStackGuard
33    pop rdx
34    pop rax
35    jmp 0x22
37    nop

```

The instruction sequence also begins with a usual stack setup. By reverse engineering the generated code, we find that the instructions from offset 0x6 to 0x21 compare the stack pointer `rsp` to the value of `kStackLimitAddressOffset`, which it accesses using a pointer to an instance of the class `v8::internal::WasmInstanceObject` stored in the `rsi` register, and if the stack pointer is not above the value of that field, the instructions from 0x2c to 0x35 call the function `WasmStackGuard` before continuing at offset 0x22.

The actual addition is performed at offset 0x22 using the “load effective address” (`leal`) instruction. However, the result is stored in the `rcx` register, whereas the

return value must be placed in `rax`, which means that the result needs to be copied from `rcx` to the `rax` register at offset `0x25` to properly return it from the function. Even in this simple example, we observe structural differences between the generated instruction sequences that contribute to the performance disparity between code produced by TurboFan and code produced by Liftoff. These structural differences include additional runtime checks and data movement in the case of Liftoff.

Interestingly, the code generated by TurboFan also accesses the `rsi` register, but does not actually use its value. In both cases, V8 uses the register to retain a pointer to a control structure. This means that the `rsi` register, which is considered a “general-purpose register” within the x64 architecture, is reserved by V8 and cannot be used as a general-purpose register by the generated code. Similar observations were made by Jangda et al. [28], who found increased register pressure to be one of the reasons why WebAssembly is slower than native code.

4.3.2 Example with Memory Access

As a second example, the following WebAssembly module exposes a single function, which loads exactly one byte from its linear memory, based on an offset given by the caller:

```
(module
  (memory 1)
  (func $load_byte (export "load_byte")
    (param $ptr i32)
    (result i32)
    (return (i32.load8_u (local.get $ptr))))))
```

The `memory` declaration specifies that each instance of this WebAssembly module requires at least one memory page. Since memory pages have a size of 64 KiB in WebAssembly, this means that the function `load_byte` is guaranteed to be able to access the first 65536 bytes. However, WebAssembly memory can be grown dynamically, and the compiler can, therefore, not statically restrict the function to that memory range. In other words, it cannot produce code that emits an error if the value of `$ptr` is larger than or equal to 65536. Instead, the generated code needs to dynamically ensure that it does not allow out-of-bounds access to memory.

TurboFan produces this instruction sequence for the function `load_byte`:

```
0    push rbp
1    movq rbp, rsp
4    push 0xa
6    push rsi
7    movq rbx, [rsi+0x17]
b    movl rdx, rax
d    movzbl rax, [rbx+rdx*1]
11   movq rsp, rbp
14   pop rbp
15   retl
16   call ThrowWasmTrapMemOutOfBounds
1b   nop
```

After the usual stack setup, the instruction at offset `0x07` retrieves a pointer to the start of the linear memory section of the WebAssembly instance, and stores it in the `rbx` register. After copying the value of `$ptr` to the `rdx` register at offset `0x0b`, the

`movzxb1` instruction at offset `0x0d` sets the return value to the zero-extended byte at the memory address `rbx+rdx`. The instructions at offset `0x11` to `0x15` merely return control to the caller.

It appears as if offset `0x16` is unreachable, and that no code was generated that ensures that `rbx+rdx` is indeed a valid pointer into the linear memory address space. While not having such protective measures in place would diminish the isolation features of WebAssembly, adding code to check the address against boundaries would require more instructions, and retrieving another value from memory, which could lead to substantially slower execution of WebAssembly functions that access linear memory.

V8 solves this problem by reserving a large virtual address range for the linear memory section of each WebAssembly instance, but only allocating physical memory for the address range that should be valid. For example, if the linear memory associated with a WebAssembly instance consists of three pages (measured in WebAssembly page sizes, i.e., 64 KiB each), and the host system uses a typical 4 KiB page size, V8 would allocate 48 pages at the beginning of the reserved virtual address range. If the linear memory is later grown, V8 simply allocates more pages within the existing virtual address range. If the process accesses memory within the reserved address range that is not backed by allocated memory, the CPU will interrupt the thread with an exception (also known as “trap” or “fault”). V8 maintains an internal mapping from the addresses of instructions that can potentially cause faults to the addresses of instructions that can handle such faults. When a fault occurs, V8 checks whether such a mapping exists for the instruction that caused the fault. In this case, V8 would find a mapping of the form `0x0d` \mapsto `0x16`, meaning that an out-of-bounds access due

to an invalid value of `$ptr` would lead to a call to `ThrowWasmTrapMemOutOfBounds`, which does not return.

This mechanism allows V8 to properly handle out-of-bounds memory accesses without any explicit boundary checks in the generated code. To ensure that this method works as intended, the address range reserved by V8 must contain every possible WebAssembly memory address. Conveniently, the WebAssembly specification restricts memory addresses to unsigned 32-bit integers, meaning that only 2^{32} bytes (4 GiB) can be addressed, relative to the beginning of the linear memory. Therefore, it is sufficient to reserve a 32-bit address range, which is tiny compared to the typical 64-bit address range available on modern hardware. Any memory operation from WebAssembly will then access some address in this reserved virtual address range, and will either hit an allocated memory page, or cause an exception.

For the same function `load_byte`, Liftoff produces this instruction sequence:

```
0    push rbp
1    movq rbp, rsp
4    push 0xa
6    subq rsp, 0x10
d    movq [rbp-0x10], rsi
11   movq rcx, [rbp-0x10]
15   movq rcx, [rcx+0x2f]
19   cmpq rsp, [rcx]
1c   jna 0x35
22   movq rcx, [rbp-0x10]
26   movq rcx, [rcx+0x17]
```

```

2a    movzxb1 rdx, [rcx+rax*1]
2e    movl rax, rdx
30    movq rsp, rbp
33    pop rbp
34    retl
35    push rax
36    call WasmStackGuard
3b    pop rax
3c    jmp 0x22
3e    call ThrowWasmTrapMemOutOfBounds
43    nop

```

The inner workings are very similar to the code generated by TurboFan. The prelude up to and including offset `0x21` was already explained above for the `add2` function. Retrieving the pointer to the start of the linear memory section requires two memory accesses at offsets `0x22` and `0x26`, after which it is stored in the `rcx` register. The `movzxb1` instruction at offset `0x2a` loads the byte from linear memory, and the next instruction copies the value to the return value, followed by a normal return sequence. The instructions from `0x35` to `0x3c` are the call to the known `WasmStackGuard` function (see above). Finally, the `call` instruction at offset `0x3e` is the fault handler for the mapping `0x2a` \mapsto `0x3e`.

4.3.3 PolyBench/C

When compiled to WebAssembly modules, all instructions are part of functions.¹ By disassembling the code generated by the WebAssembly compilers Liftoff and TurboFan, it is possible to analyze the number and type of the instructions.

4.3.3.1 Instruction Count and Branch Instructions

Figure 4.3 shows the total number of CPU instructions generated for each PolyBench/C benchmark by each compiler. As expected, Liftoff generates significantly more instructions than TurboFan.

Similarly, as depicted in Figure 4.4, Liftoff also generates more branch instructions than TurboFan, where branch instructions are any instructions that change the control flow (`call`, `ret`, and conditional and unconditional jump instructions). However, Figure 4.5 shows that about 80% of the generated `call` instructions produced by both compilers are only used for error handling, similar to the use of `call ThrowWasmTrapMemOutOfBounds` in Section 4.3.1. Therefore, it makes sense to only consider jump instructions, both conditional and unconditional, to compare the generated code. This statistic is depicted in Figure 4.6. The difference is even larger here, with Liftoff producing far more jump instructions than TurboFan. This can potentially make branch prediction within the system’s CPU less effective [55, 56].

¹The WebAssembly specification permits instructions outside of functions, however, these instructions can only produce constant values, and are, therefore, not relevant for this analysis.

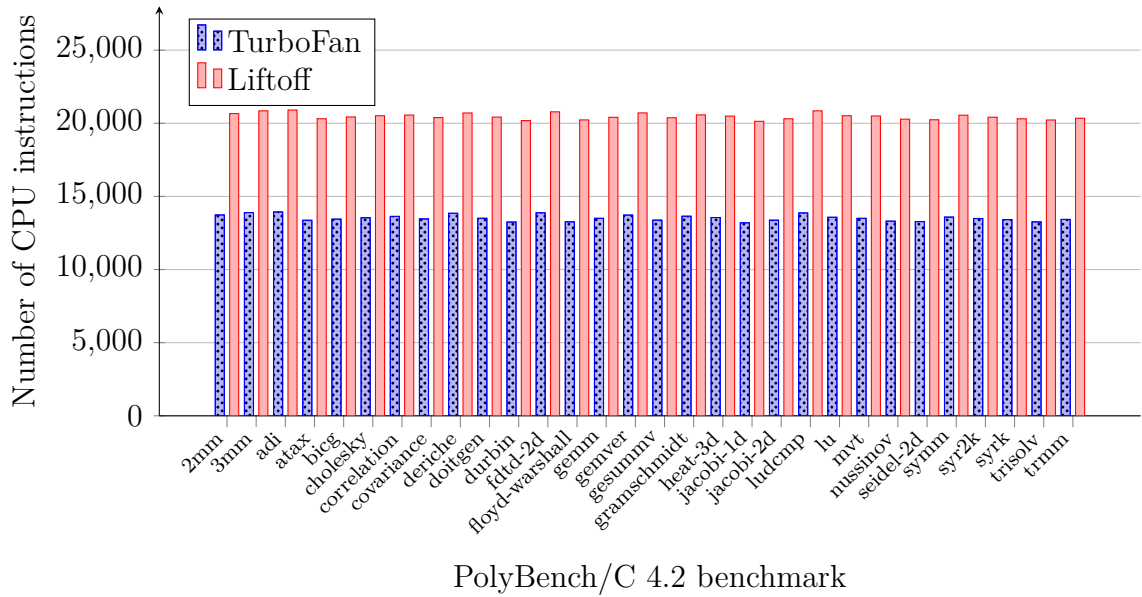


Figure 4.3: Number of CPU instructions generated by each compiler for each of the PolyBench/C benchmarks

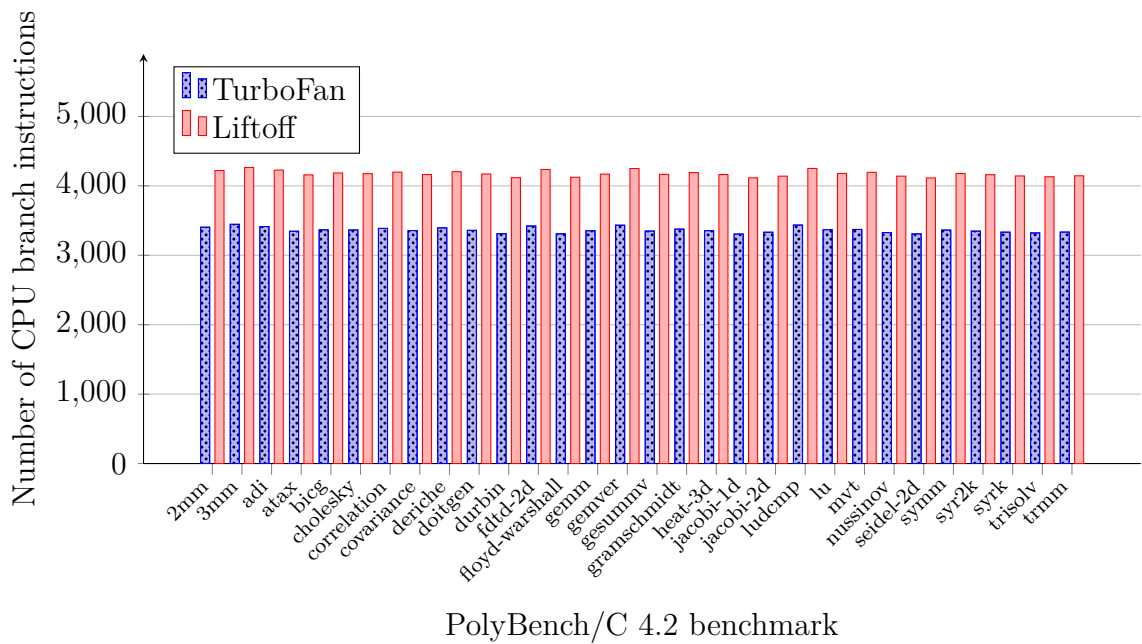


Figure 4.4: Number of CPU branch instructions generated by each compiler for each of the PolyBench/C benchmarks

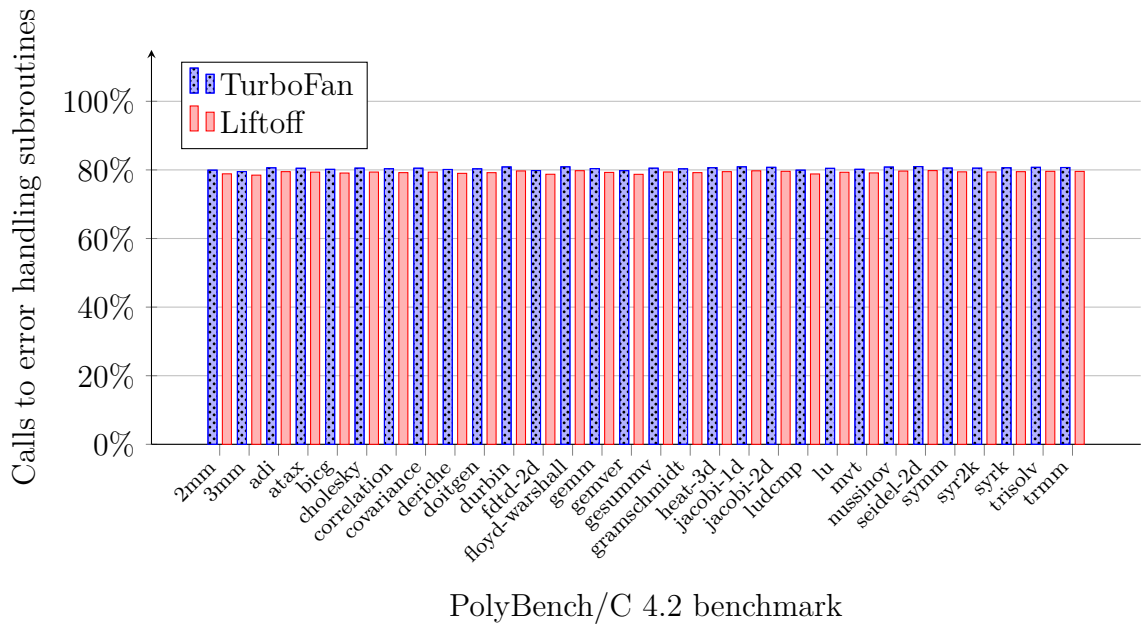


Figure 4.5: Percentage of calls to error handlers with respect to all call instructions generated by each compiler for each of the PolyBench/C benchmarks

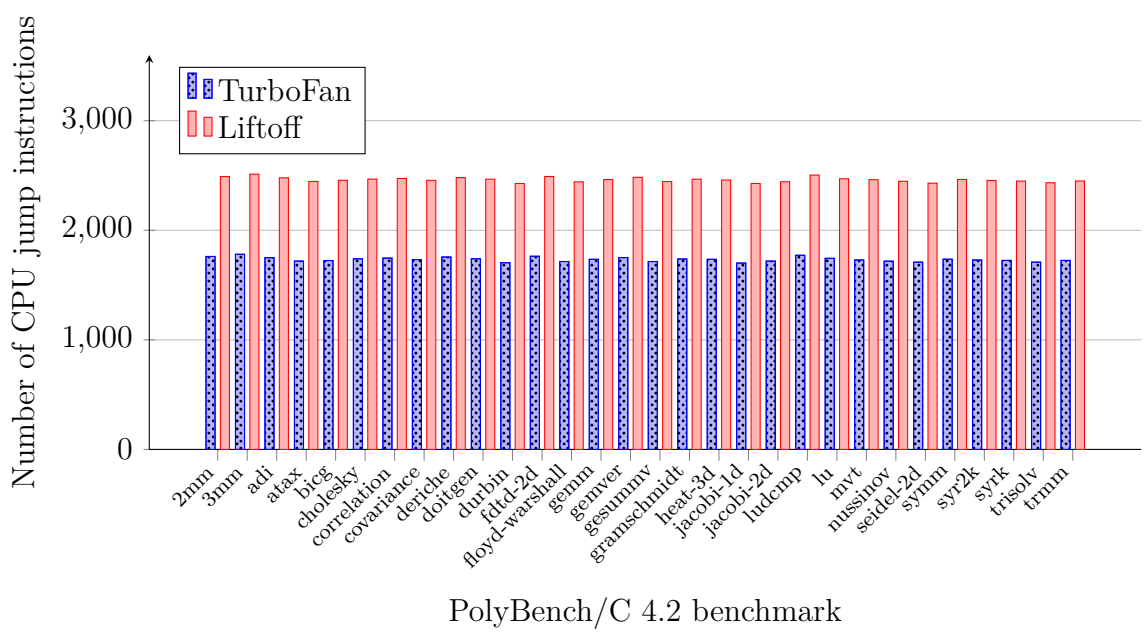


Figure 4.6: Number of CPU jump instructions generated by each compiler for each of the PolyBench/C benchmarks

4.3.3.2 Memory Access

However, the number of instructions alone is not the only problem that we can identify based on a simple statistical analysis. Since the PolyBench/C benchmarks do not perform system calls, especially no I/O operations, their performance mostly depends on the execution of their respective instructions by the system's CPU. While operations on CPU registers are particularly fast, access to main memory is relatively slow. The primary reasons for memory access are threefold:

1. In order to execute instructions, the processor needs to load the instructions themselves from memory. Liftoff has two disadvantages with respect to this aspect. First, since it generates more instructions than TurboFan to achieve the same goal, the processor needs to load more instructions from memory, and loading more data from memory is generally slower than loading less. For the second problem, it is important to know that (consecutive) instructions will usually be served from the processor's L1 instruction cache after being loaded from main memory once, and even though the compiler can greatly influence how much the generated code benefits from said cache [24], modern processors are capable of predicting control flow to fetch instructions early enough to avoid delays [55]. However, if the control flow changes in complex or unpredictable patterns, the processor might fail to predict it in time, meaning that required instructions might not be in the processor's cache, and could, therefore, delay program execution substantially [56]. This becomes more likely with an increasing number of branch instructions, because they make it more

difficult for the processor to predict the control flow, and, as we have seen above, Liftoff generates significantly more branch instructions.

2. Some instructions, such as `push`, `pop`, and `stos`, access main memory due to their nature. While `push` and `pop` operate on the current stack frame, which is likely cached within the processor, other instructions, including `stos`, can access any virtual memory address, and can, therefore, be much slower.
3. Most CPU instructions take one or more operands, for example, a `mov` instruction copies data from a source operand to a destination operand. In many such cases, the instruction set allows either operand to be a memory location. Thus, the processor might need to access main memory depending on the instruction's operands.

We cannot fully assess the effects of the first reason through static analysis, but we can analyze the code with respect to the other two reasons for memory access.

Figure 4.7 shows the percentage of CPU instructions that access main memory for each benchmark and for each compiler. When compiled using Liftoff, 49.9% of the generated CPU instructions access main memory, compared to only 36.8% when using TurboFan.

To demonstrate that this is not solely caused by TurboFan selecting different instructions, we created the same statistic for generated `mov` instructions only, which simply copy a single value from a source register or memory location to a destination register or memory location. Figure 4.8 shows that, even when only considering these instructions, Liftoff relies significantly more on main memory than TurboFan.

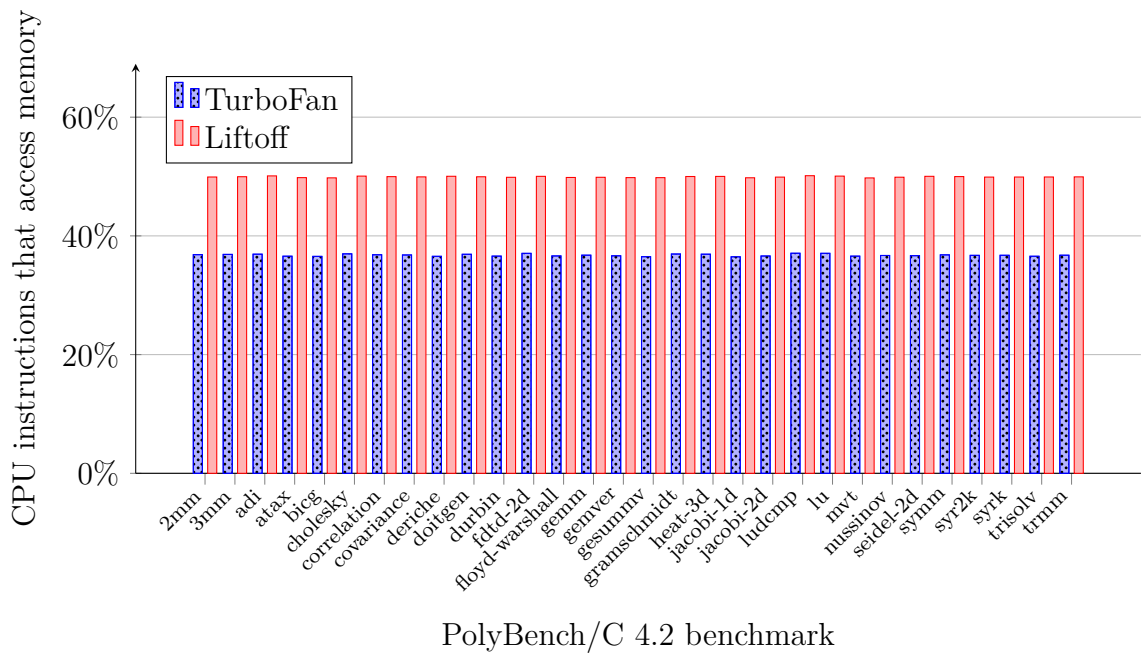


Figure 4.7: Percentage of CPU instructions that access main memory

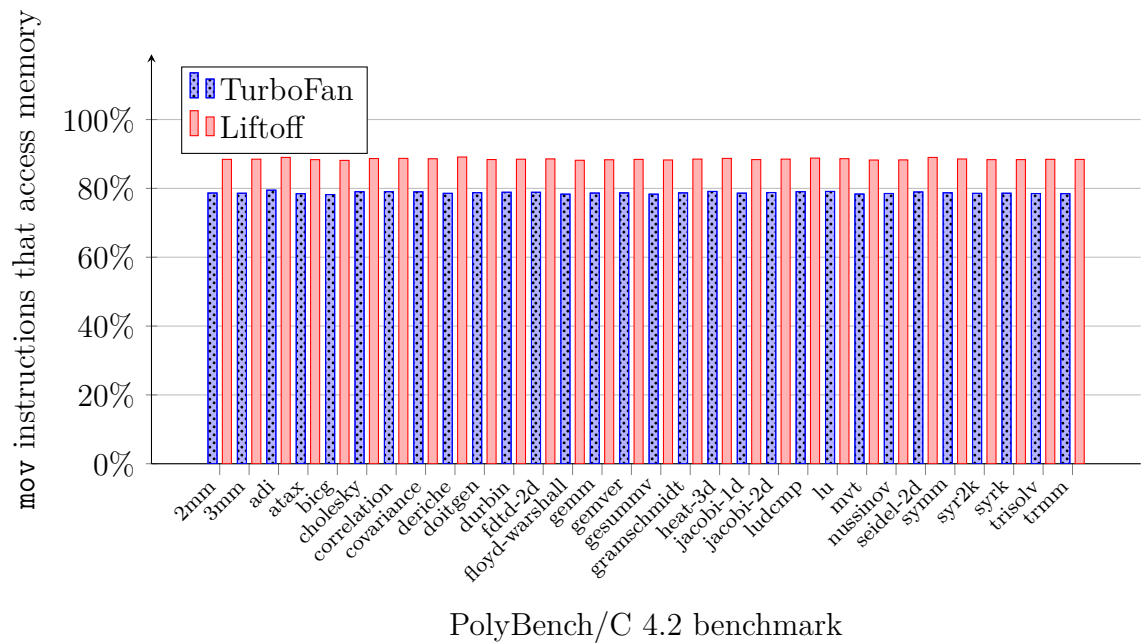


Figure 4.8: Percentage of CPU mov instructions that access main memory

In conclusion, we have seen that, as expected, Liftoff produces more instructions, more branch instructions, more jump instructions, and more instructions that access memory (as compared to instructions that only access registers). Overall, these differences explain large performance differences between the code generated by TurboFan and the code generated by Liftoff. This shows that TurboFan is more effective than Liftoff. The next chapter will consider the efficiency of the two compilers.

Chapter 5

Compiled Code Caching

Due to portability and security concerns, WebAssembly was designed to be compiled to the target architecture's instruction set at runtime (see Chapter 4). However, when running code from a trusted source on a single architecture, or untrusted code within a container or sandbox, these concerns become less relevant. Especially in scenarios where a Node.js application is expected to be initialized quickly, for example, when used as a command-line tool, as a desktop application, or in serverless computing, performance might be a more crucial factor. Here, using WebAssembly modules by first compiling them can cause visible delays.

A possible solution to performance problems caused by extensive computations, such as compilation, is *caching*. The first time the computation result is required, it is computed, and the result is stored in a persistent manner. Subsequent requests for the result of the same computation can then be fulfilled with the stored result, and do not require the result to be computed again. However, it is worth noting that the additional logic and data transfer can also introduce a non-trivial overhead.

For WebAssembly compilation, caching means storing the compilation result, i.e., generated instruction sequences and associated information (see Chapter 4), and restoring them when the same WebAssembly code is compiled at a later time. This does not eliminate the need for compilation entirely because the WebAssembly code still needs to be compiled at least once, but it removes the need for compilation in subsequently started processes or threads that use the same WebAssembly modules. In this chapter, we will demonstrate the feasibility of caching compiled WebAssembly code and analyze the performance improvements enabled by caching. Further, we will design, implement, and experimentally evaluate a shared compiled code cache for WebAssembly code in Node.js applications.

5.1 Code Extraction and Insertion

Prior to designing a shared code cache, we need to find a way to efficiently retrieve compiled code from V8, and later inject the same code in a different V8 process.

While current versions of V8 provide such features for streaming WebAssembly compilation, no usable interface exists for Node.js, which only supports non-streaming WebAssembly compilation (see Section 3.5.1). However, V8 has internal functions that allow serializing compiled WebAssembly modules into byte sequences, and deserializing byte sequences into compiled WebAssembly modules. These byte sequences contain the generated instruction sequences and associated information required to use them.

We developed a native add-on for Node.js that exposes these internal V8 features to Node.js applications through two functions:

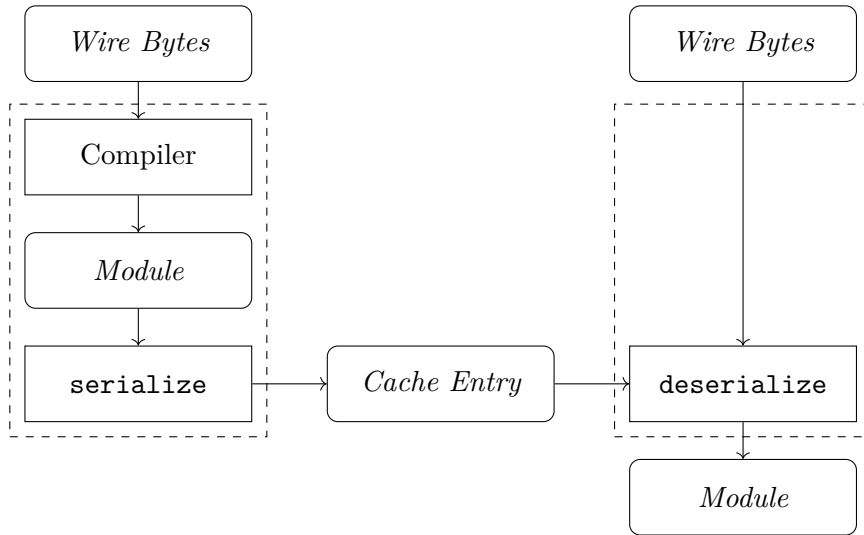


Figure 5.1: WebAssembly cache data flow: Cache entry creation (left) and cache entry retrieval (right)

1. `serialize(m)` returns a byte sequence as a JavaScript `ArrayBuffer` [13] based on a given `WebAssembly.Module` [67].
2. `deserialize(b, s)` creates an instance of the `WebAssembly.Module` [67] class based on the WebAssembly module bytes `b` (referred to as “wire bytes” within V8) and the byte sequence `s`, which was generated by the `serialize` function.

This pair of functions is sufficient to extract code from a compiled WebAssembly module, store it in a cache entry, and later use the cache entry to obtain a usable module. This data flow is depicted in Figure 5.1.

V8 allows selectively disabling Liftoff and TurboFan. If a process is started with only Liftoff enabled, V8 prevents inserting code generated by TurboFan (and vice versa). Similarly, if a specific compiler feature is enabled (or disabled) in one process, the code generated within that process cannot be loaded in another process in which the same compiler feature is disabled (or enabled, respectively). A proper

cache lookup therefore requires knowledge about the current process’s V8 configuration. To achieve this, our Node.js add-on allows applications to check relevant V8 configuration flags.

In order to create realistic benchmarks, we extracted 115 WebAssembly modules from existing JavaScript applications, with module sizes ranging from as little as 1068 bytes to 37.3 MiB. This was possible by capturing, deciphering, and decoding network traffic from web browsers through which a user accessed various web applications, and by extracting WebAssembly modules from the decoded data streams.¹ It would be difficult to run the code represented by the WebAssembly modules in the way intended by their creators, since each module performs application-specific tasks and has certain requirements towards its host environment. However, the experiment in the following sections is focused on compiling WebAssembly modules, which does not require running the compiled modules. Other experiments, which will be described in Section 5.3.4.1 and Section 5.3.4.2, will use a different set of WebAssembly modules, which can be executed.

The approach Park et al. [47] used to cache compiled JavaScript code used cache entries that were much larger than the original JavaScript files. Similarly, we observe that serialized compiled WebAssembly modules are often considerably larger than the original WebAssembly files. Figure 5.2 shows the ratio of the serialized size to the original size based on the WebAssembly modules we extracted from existing applications, depending on which compiler was used by V8. In the case of

¹We obtained a small number of WebAssembly modules that could not be compiled without non-standard compiler configurations in the Node.js version that we used in our experiments. These modules are, therefore, not included in the set of 115 WebAssembly modules that is described here. (See Section 6.1.)

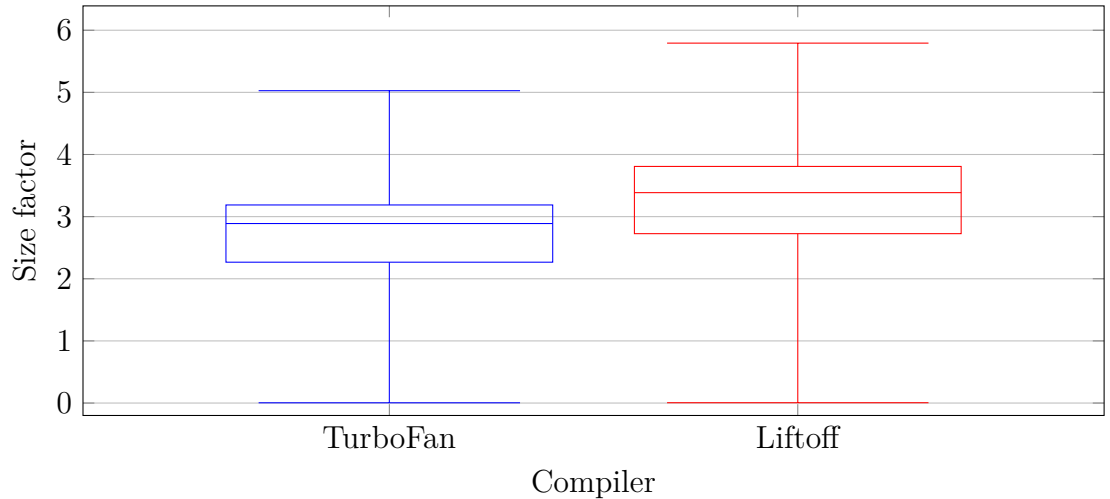


Figure 5.2: Ratio of serialized compilation result size to WebAssembly module size

JavaScript, better performance was achieved by caching optimized code in addition to intermediate bytecode, effectively increasing the size of cache entries [47]. For WebAssembly, it is sufficient to store the optimized compilation result (left side), which is significantly smaller than the result of the non-optimizing compiler Liftoff (right side), but still up to five times larger than the original WebAssembly module. The lower limit for both ratios appears to be zero in Figure 5.2. This is caused by one of the WebAssembly modules containing almost no code, but a large amount of debugging information, which is ignored by the WebAssembly compiler. Due to the small number of instructions, the serialized compilation result is tiny compared to the original WebAssembly module. To better demonstrate this, the `wasm-strip` tool [71] can be applied to all WebAssembly modules, which removes optional information, such as debugging information, from binaries. This modification reduces the sizes of the WebAssembly modules, but it does not affect the compilation results or their sizes. The updated ratios are shown in Figure 5.3 and clearly differ from zero.

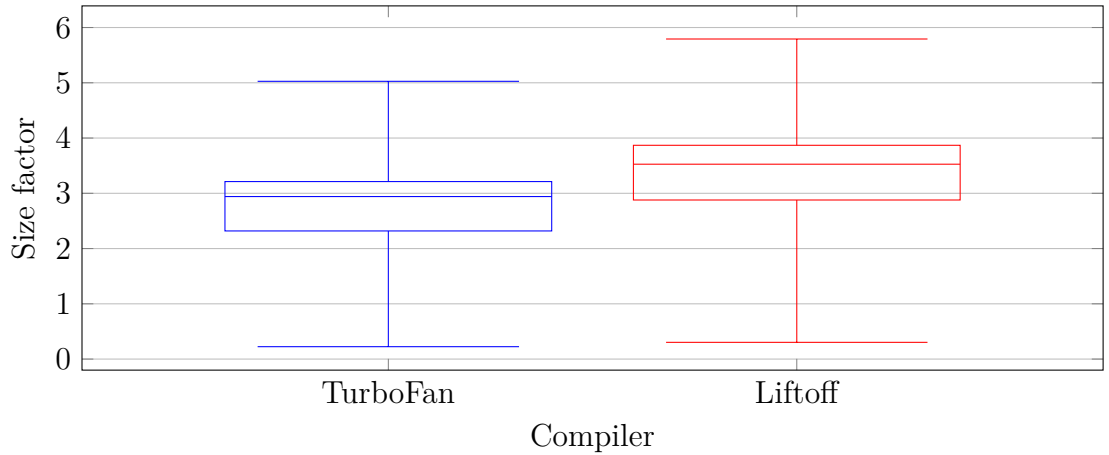


Figure 5.3: Ratio of serialized compilation result size to WebAssembly module size after applying `wasm-strip`

5.2 Compiler Performance versus Code Caching

Most importantly, we need to compare the performance of both compilers, Liftoff and TurboFan, to the previously described deserialization method. The hypothesis is that deserializing cached code uses fewer resources than compilation.

In order to test this hypothesis, we measured the real elapsed time it takes to obtain a compiled, usable module from the WebAssembly module bytes (“wire bytes”) through compilation. To achieve comparable results, we disabled tiering-up (see Section 3.5.1), only enabled one compiler at a time, and used the synchronous `WebAssembly.Module` constructor for Liftoff and TurboFan. Additionally, we took the same measurements for the deserialization method. In this case, the module had already been compiled and optimized by the TurboFan compiler, and the serialized compiled module is available in memory (in addition to the wire bytes).

Each measurement is taken in a separate process. For each such process, we also record the CPU time of the process, that is, the total duration that threads of the

process were scheduled on any of the CPU cores in user or system mode, and the peak physical memory usage through the `VmHWM` statistic provided by the Linux kernel. These metrics are equally if not more important than the elapsed real time required to compile a WebAssembly module. Even under the simplified assumption that CPU time and (physical) memory are the only resource constraints of a process, both of these resources are finite, and must be shared among all processes on the same system. While a high CPU time to real elapsed time ratio is an indication of well-designed parallelism, it also means that few concurrent instances of the same process might already use all available CPU time, and any additional instances could cause the performance of all processes to degrade. For cloud applications, it is realistic to assume that more than one process will be active on the same hardware at a time. We recorded each measurement for each of the 115 WebAssembly modules 100 times. The mean values of elapsed real time, CPU time, and memory usage for each module are depicted in Figures 5.4, 5.5, and 5.6, respectively. As shown in Figure 5.4, all three methods generally take longer for larger modules than for smaller ones, which is unsurprising because larger modules generally contain more code that needs to be compiled or loaded.

For legibility, Figures 5.4, 5.5, and 5.6 do not include error bars. Instead, Figures 5.7, 5.8, and 5.9 show the significance of improvements. For two variables with mean values μ_1, μ_2 and standard deviations σ_1, σ_2 , we define the *significance* of the change as $(\mu_1 - \mu_2)/(\sigma_1 + \sigma_2)$. By convention, we call the difference *statistically significant* if the significance is at least one. In this case, the probability that a random sample of either variable could have been obtained by sampling the other variable is strictly less than 5 %.

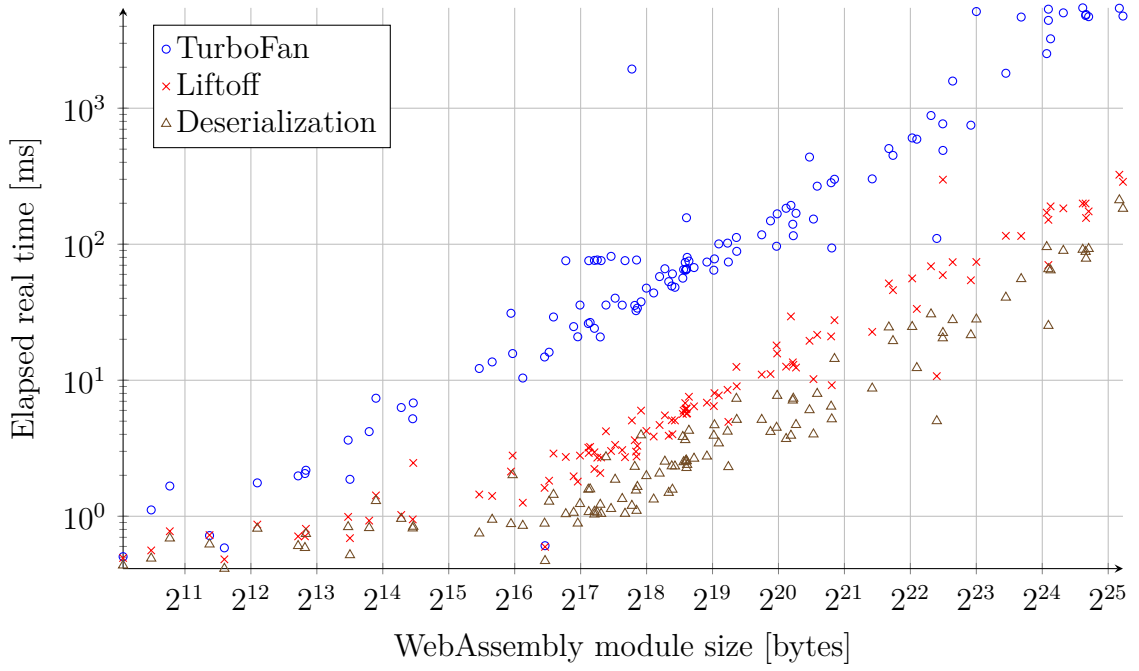


Figure 5.4: Compilation times by approach

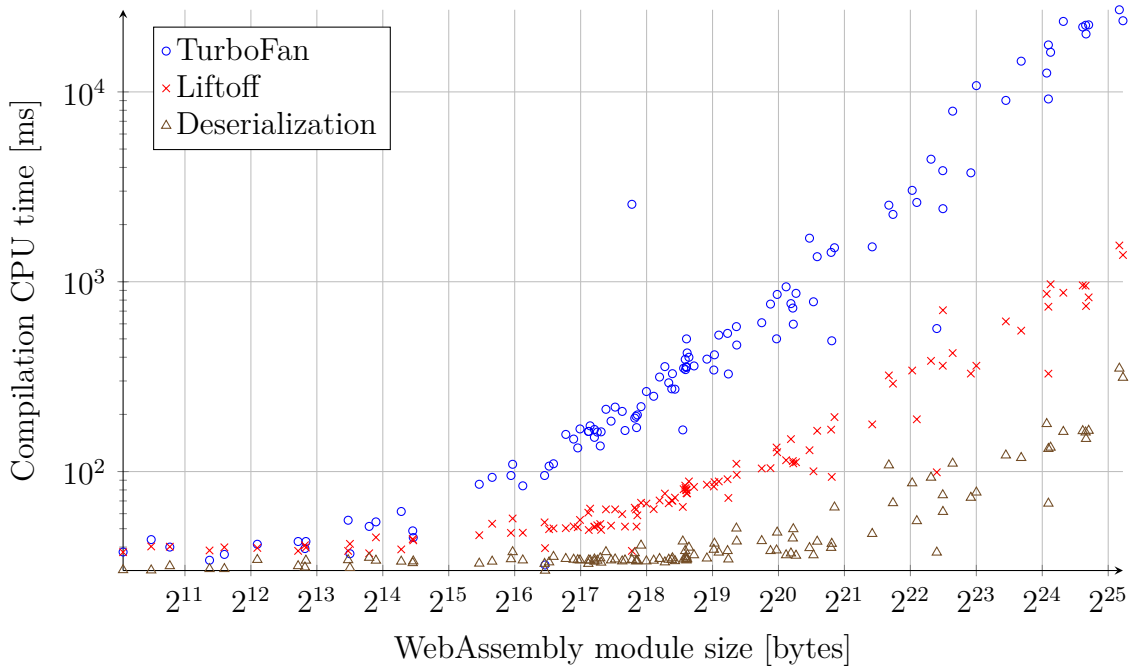


Figure 5.5: Compilation CPU times by approach

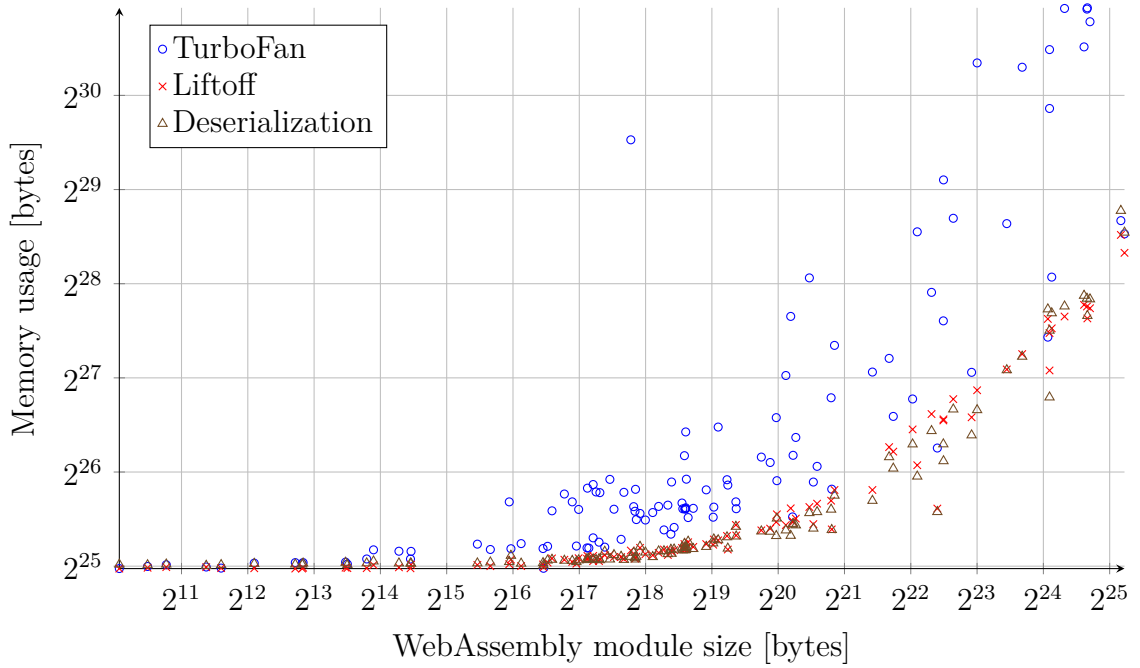


Figure 5.6: Compilation memory usage by approach

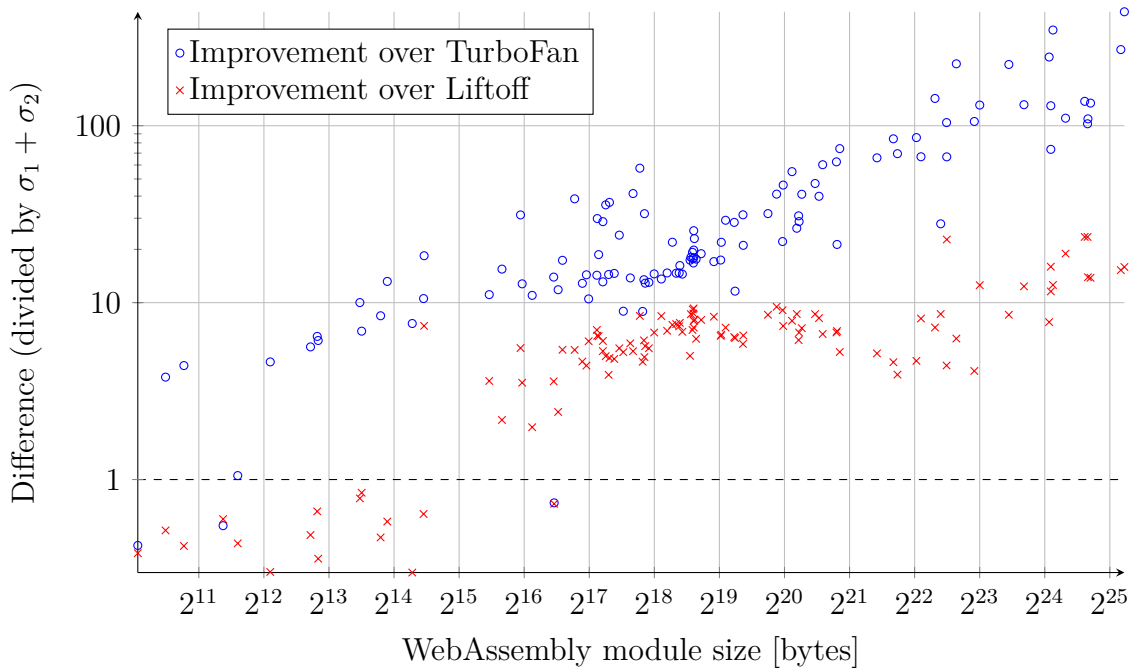


Figure 5.7: Significance of compilation time improvements

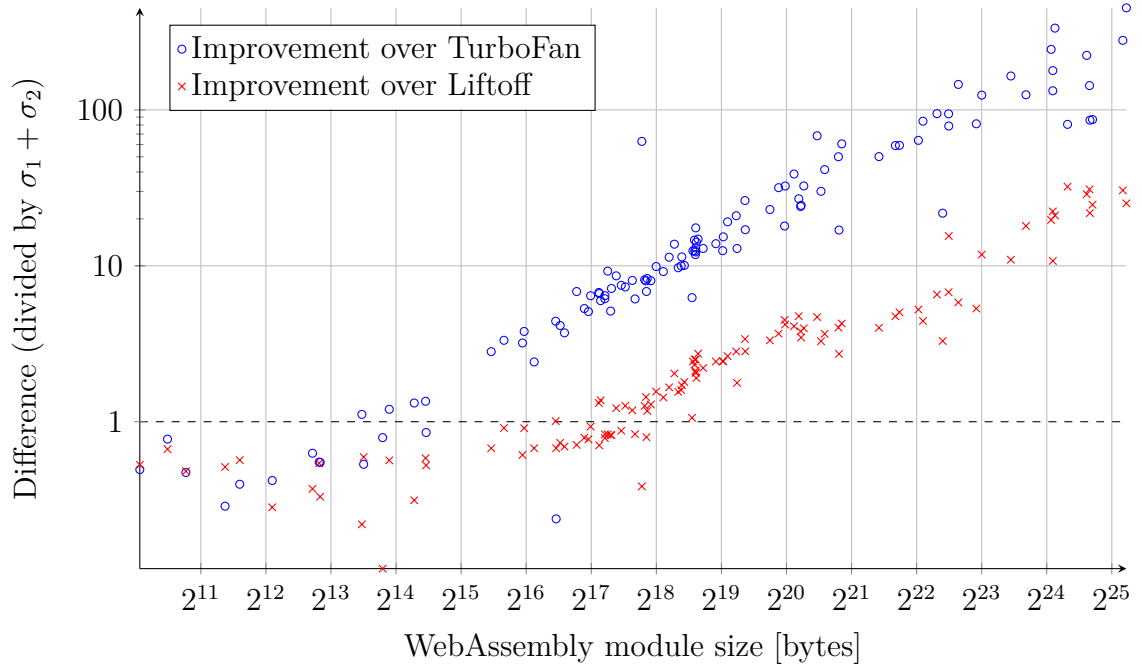


Figure 5.8: Significance of compilation CPU time improvements

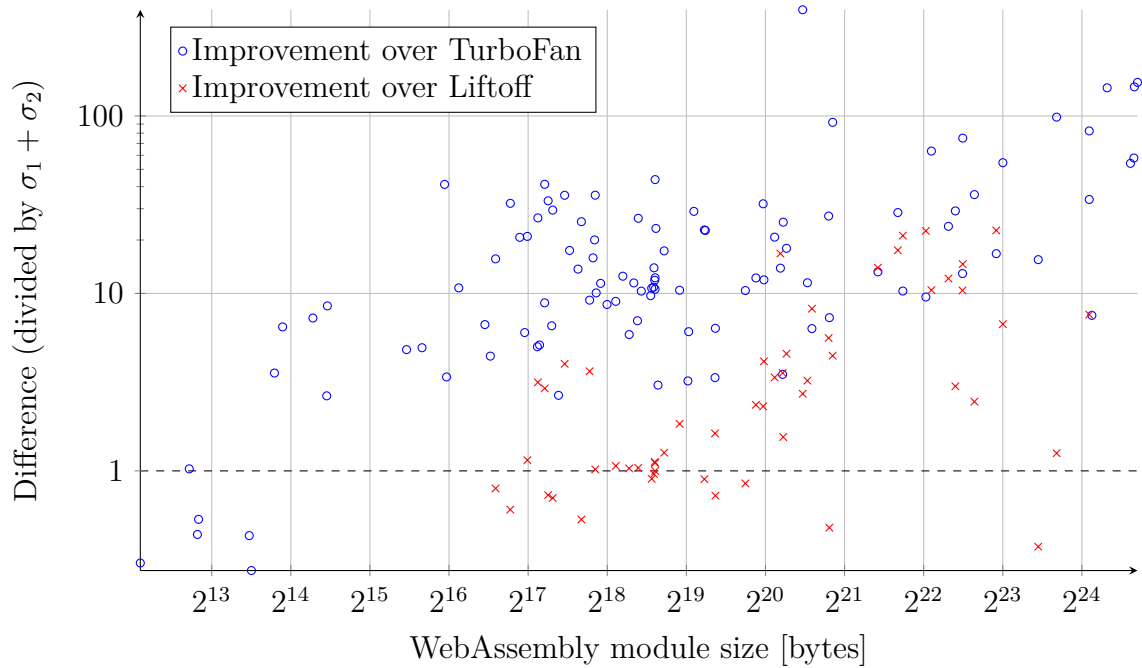


Figure 5.9: Significance of memory usage improvements

By this definition, Liftoff was significantly faster than TurboFan for 111 modules (96.5%), used significantly less CPU time for 98 modules (85.2%), and had a significantly smaller memory footprint for 110 modules (95.7%).

When comparing deserialization to compilation using TurboFan, we measured statistically significant compilation time improvements for 112 modules (97.4%), CPU time improvements for 102 modules (88.7%), and memory usage improvements for 101 modules (87.8%). For 111 modules (96.5%), the speedup was at least two, and for 77 modules (67.0%), the speedup was at least 20. Similarly, for 98 modules (85.2%), the CPU time was reduced by at least 50%, and for 54 modules (47.0%), it was reduced by at least 90%.

Compared to compilation using Liftoff, we observed significant compilation time improvements for 99 modules (97.4%), significant CPU time improvements for 77 modules (67.0%), and significant memory usage improvements for 41 modules (35.7%). For 69 modules (60.0%), the speedup was at least two. The CPU time was reduced by at least 50% for 64 modules (55.7%).

The only statistically significant regression is an increase in memory usage for 39 modules (33.9%) when compared to Liftoff, and for six modules (5.2%) when compared to TurboFan. In these cases, however, the difference is small (less than 20%, see Figure 5.6).

Since the deserializer is synchronous, it is consistent to compare it to synchronous WebAssembly compilation. However, we also repeated this experiment with asynchronous WebAssembly compilation, and found that asynchronous compilation was significantly slower than synchronous compilation for 85 modules (73.9%) in the case of TurboFan, and for 91 modules (79.1%) in the case of Liftoff. For both compil-

ers, this results in even larger differences when compared to deserialization, and we, therefore, decided not to present these results in detail.

We also repeated the experiment with deserialization of code generated by Liftoff instead of optimized code produced by TurboFan, which, as discussed in Section 5.1, leads to a larger serialized format. We found that it also causes longer deserialization times, and no significant improvements of real elapsed time, CPU time, or memory usage. Therefore, this data is also not presented in detail at this point

In conclusion, compiled code caching can drastically reduce compilation times as well as CPU and memory usage during compilation.

5.3 Shared Code Cache

With the previously discussed cache creation and retrieval method from Section 5.1 and the performance benefits of code caching presented in Section 5.2, it is viable to construct a disk-based cache that can be populated and used by individual processes. However, this approach is troublesome for large application clusters: First, write access to the code cache should be controlled strictly to prevent malicious code injections, and it might be undesirable for all processes that use WebAssembly to have permission to write compiled code to the cache. Second, if a process uses Liftoff or tiering-up to improve its own startup time (see Section 5.2), it might not insert optimized code into the cache, but instead the output of the baseline compiler. Third, a disk-based cache might reduce expected speedups due to disk access times associated with potentially large cache entries (see Section 5.1), which each new process might copy from disk into memory. Synchronization between processes is

also a potential issue, for example, a safe cache design must make it impossible for processes to access partially created cache entries.

5.3.1 WebAssembly Module Identification

The client needs to be able to identify each WebAssembly module in order to look it up in a cache. Since the WebAssembly JavaScript Interface is agnostic to the source of the WebAssembly module code (see Section 3.5.1), a module can only be identified by its code, and no file path or URL is available. This is an important difference to other runtimes, e.g., Java [46], where classes can be identified by a unique name, and to traditional compilers that rely on file paths to identify compilation units.

In other words, a fingerprinting function is required that maps the byte sequence representing a WebAssembly module (“wire bytes”) to an identifier. The simplest choice for such a fingerprinting function is the mathematical identity function, meaning that the fingerprint of a module would be the module bytes themselves. However, we have found numerous WebAssembly modules with sizes of several megabytes, and using such large identifiers of varying length to identify cache entries is inefficient. We therefore consider better alternatives for the choice of the fingerprinting function that result in short and fixed-length fingerprints.

A variety of linear-time functions exist that map large amounts of in-memory data to short identifiers. Traditional information-theoretic algorithms, such as CRC32C, and cryptographic hash functions, such as SHA-1, provide reliable and, in the case of hash functions, collision-resistant identification methods, and run in $\Theta(n)$. Collision resistance of a cryptographic hash function f means that it is infeasible for a potential

attacker to produce a collision, that is, two WebAssembly modules x and y for which $f(x) = f(y)$ holds. If the selected function is not collision-resistant, an attacker could potentially cause a collision, and, therefore, inject undesired or malicious code into the shared code cache if they can invoke the WebAssembly compilation procedure. Choosing a cryptographic hash function as the fingerprinting function mitigates this risk due to its collision resistance.

The disadvantage of linear-time hash functions is their performance impact when computing the fingerprint of large WebAssembly modules. For scenarios in which an attacker cannot cause arbitrary WebAssembly modules to be compiled and cached, we propose using constant-time hash functions. We define the function family \mathbf{fp}_r for $r \geq 2$ as follows: Let b_0, \dots, b_{n-1} be the input byte sequence, with $b_i \in \{0, \dots, 2^8 - 1\}$ for $0 \leq i < n$. Let p be a linear congruential generator with

$$\begin{aligned} p(0) &= b_{\lfloor n/2 \rfloor}, \\ p(i+1) &= (a * p(i) + c) \bmod 2^{32}. \end{aligned}$$

We chose $a = 1664525$ and $c = 1013904223$ as suggested by Knuth [30]. Let the result be the r byte vector f_0, \dots, f_{r-1} with

$$\begin{aligned} f_0 &= \lfloor n/2^8 \rfloor \bmod 2^8, \\ f_1 &= n \bmod 2^8, \\ f_{2+i} &= b_{p(i+1) \bmod n}. \end{aligned}$$

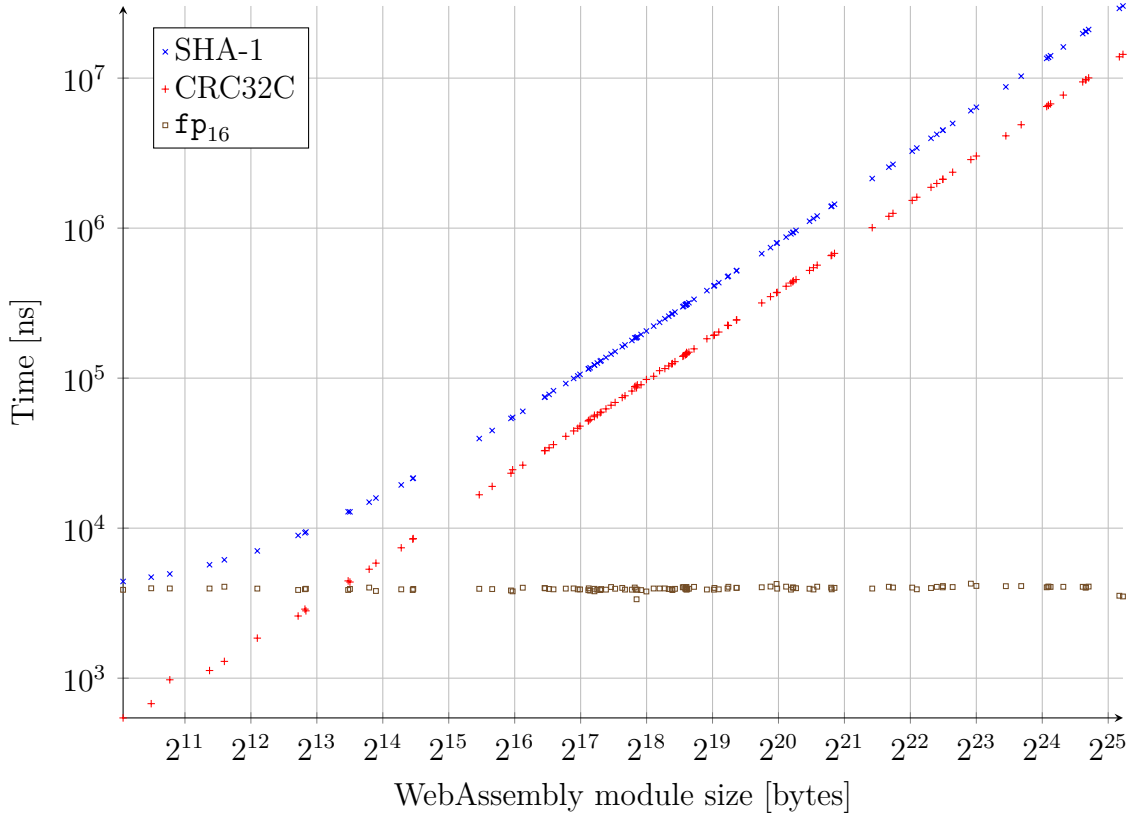


Figure 5.10: Performance comparison between `fp16`, `CRC32C`, and `SHA-1` in Node.js

In other words, the result consists of the length n modulo 2^{16} and the module bytes at $r - 2$ pseudo-random locations. Each such function fp_r , only requires 32-bit integer arithmetic, can be implemented efficiently in JavaScript, and runs in $\mathcal{O}(1)$. We used fp_{16} in our implementation. While the resulting “fingerprint” is neither unique nor (cryptographically) collision-resistant, it is sufficiently unlikely to collide with another module’s fingerprint for the purpose of a shared cache. Figure 5.10 shows a performance comparison between fp_{16} , `CRC32C`, and `SHA-1` in Node.js, based on running each function 100,000 times on each of the 115 WebAssembly modules. With a constant runtime of $4.0 \mu\text{s}$, fp_{16} is significantly faster than other methods.

The following theorem shows that no constant-time function is collision-resistant, meaning that this is not a shortcoming of this particular constant-time fingerprinting function, but rather a shortcoming of all functions that run in sub-linear time.²

Theorem. *No constant-time function f is collision-resistant.*

Proof. Let t be the finite runtime of the function f , measured in any discretization of time. Let m be the maximum number of bytes the function can access during the smallest increment of the chosen discretization of time. Let $x = (x_1, \dots, x_n)$ be any byte sequence with $n > mt$. If f is collision-resistant, then finding a byte sequence y with $y \neq x$ and $f(x) = f(y)$ is infeasible.

Since the runtime of f is t , the function can access at most mt bytes of x . Let L_x be the set of indices within x that are accessed by $f(x)$. (L_x can be computed in constant time.) Since $|L_x| \leq mt < n$, there exists an $i' \in \{1, \dots, n\} \setminus L_x$. Let $y = (y_1, \dots, y_n)$ be any byte sequence that fulfills

$$x_i = y_i \iff i \neq i'.$$

Now $x \neq y$ and $f(x) = f(y)$. □

5.3.2 Design

To circumvent these problems, we designed and implemented a novel approach to share compiled WebAssembly code between Node.js processes. In the following, we will refer to the processes of one or more Node.js applications as *client processes*. In

²The extension of the theorem from constant-time functions to arbitrary sub-linear time functions is left to the reader.

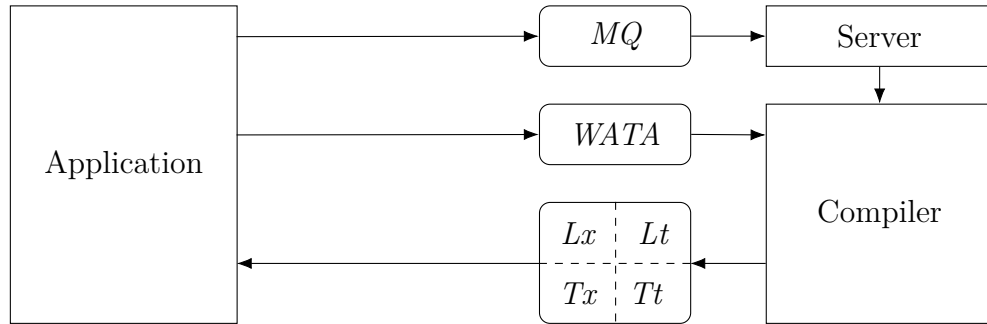


Figure 5.11: Shared cache server architecture

this context, a *V8 configuration* is a set of flags that affect V8’s internal behavior (see Section 5.1). The cache implementation prevents loading incompatible cache entries, and potentially maintains multiple cache entries for the same WebAssembly module, but for different V8 configurations. For example, Figure 5.11 presents the compiled code cache as a matrix of four configurations Lx , Lt , Tx , and Tt , where the first letter indicates which is the fastest enabled compiler (either **L**iftoff or **T**urboFan), and the second letter indicates whether the compiler is used **e**xclusively, or if **t**iering up is enabled.

5.3.2.1 Client Processes

Client processes (see Figure 5.12) compile WebAssembly modules through a modified WebAssembly JavaScript Interface, which is compatible with the one described in Section 3.5.1. This means that no changes to the application code are necessary. The compilation procedure, given a module represented by bytes b_0, \dots, b_{n-1} , computes the module identifier $\text{fp}_{16}(b_0, \dots, b_{n-1})$, and attempts to locate a cache entry in a shared memory segment based on the computed module identifier and the current V8 configuration. If such an entry exists, the compilation procedure deserializes the

cache entry to obtain a WebAssembly module instance without having to compile the module bytes. If no such entry exists, the client process copies b_0, \dots, b_{n-1} into a new shared memory segment, which we refer to as the *WebAssembly Transfer Area* (WATA), and sends a pointer to the WATA and the current V8 configuration to an existing message queue (MQ). Finally, the client process falls back to V8's original compilation procedure, which, depending on the current configuration, uses Liftoff and/or TurboFan to compile the code.

5.3.2.2 Cache Server Process

A separate server process (see Figure 5.13) is responsible for creating the message queue (MQ). Upon receiving a pointer to a WebAssembly Transfer Area (WATA) along with a valid V8 configuration through the message queue, the server process starts a new compiler process with parameters matching the received V8 configuration and passes the pointer to the WATA to the newly created compiler process.

5.3.2.3 Compiler Processes

When started by the cache server process, the compiler process (see Figure 5.14) loads the module bytes b_0, \dots, b_{n-1} from the WATA and computes the module's fingerprint $\text{fp}_{16}(b_0, \dots, b_{n-1})$. After ensuring that no other compiler process is already compiling the same module with the same V8 configuration, the process compiles the module. If TurboFan has not been disabled in the given V8 configuration, the compiler process uses it to produce optimized code. Only if TurboFan has been disabled, the compiler process uses Liftoff, and therefore generates unoptimized code. Once compilation finishes, the compiler process serializes the compiled WebAssembly module, and

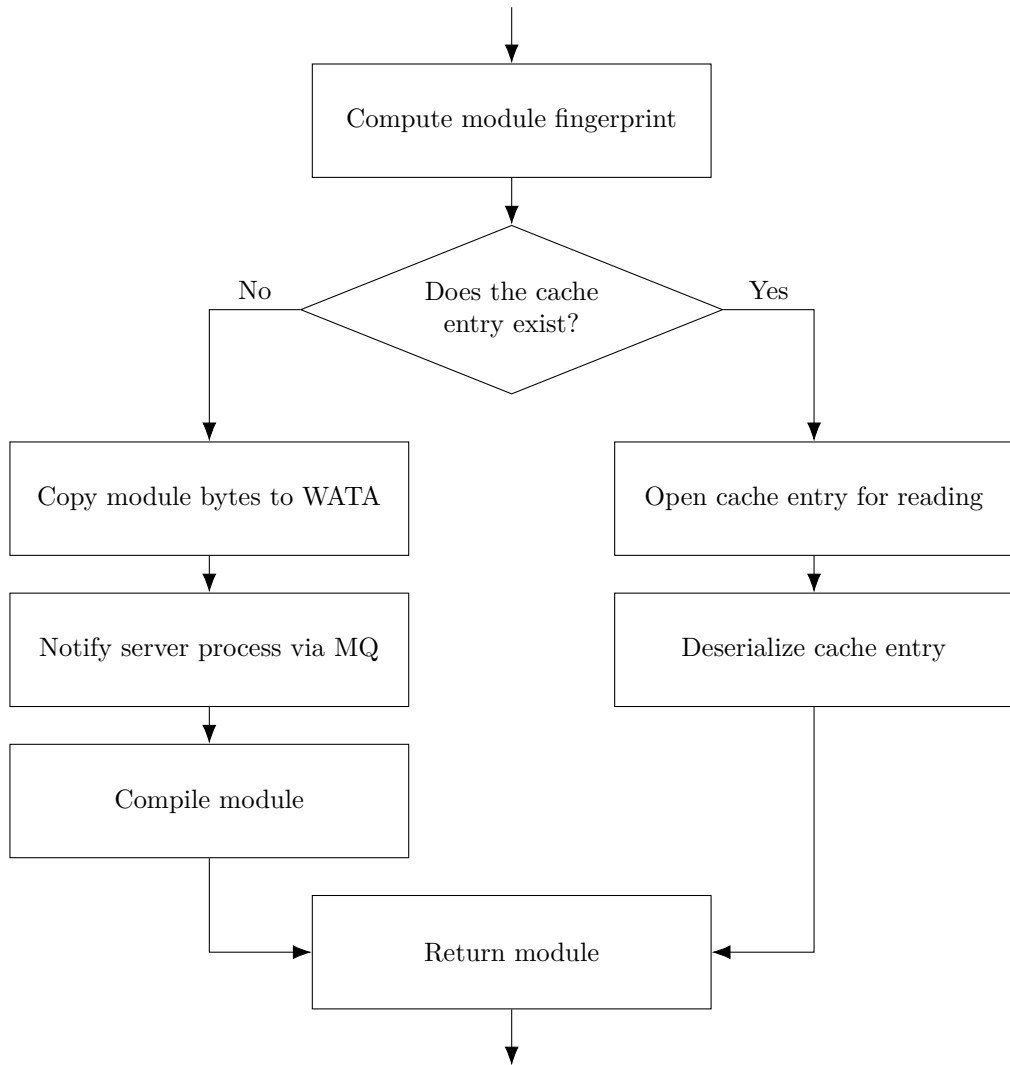


Figure 5.12: Cache design: Client process control flow

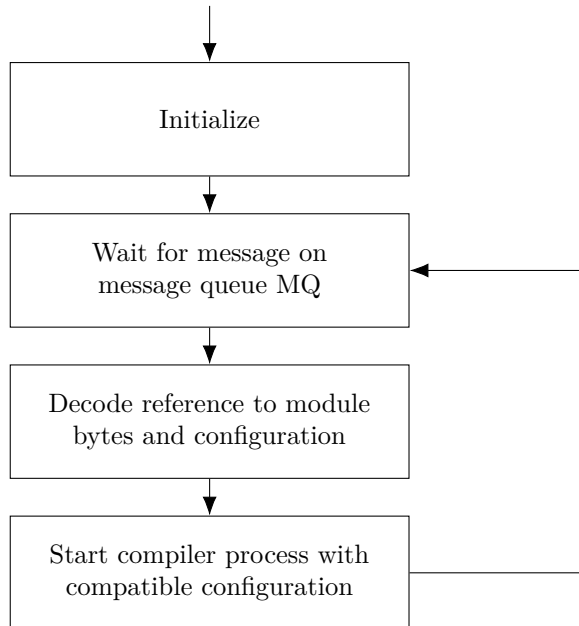


Figure 5.13: Cache design: Server process control flow

writes the result to the shared code cache in shared memory. In any case, it releases the WATA by deleting the reference to the module bytes, causing the underlying memory to be freed, before terminating.

5.3.3 Implementation

The modified WebAssembly JavaScript Interface was implemented in JavaScript, except for the deserialization logic, which was implemented in C++ due to the necessity to access internal V8 features (see Section 5.1), and communication with the shared cache server’s message queue, which was implemented in C++ and uses POSIX functions to write to the message queue.

Similarly, the server process code is written in C and uses low-level POSIX functions to receive messages on the message queue MQ.

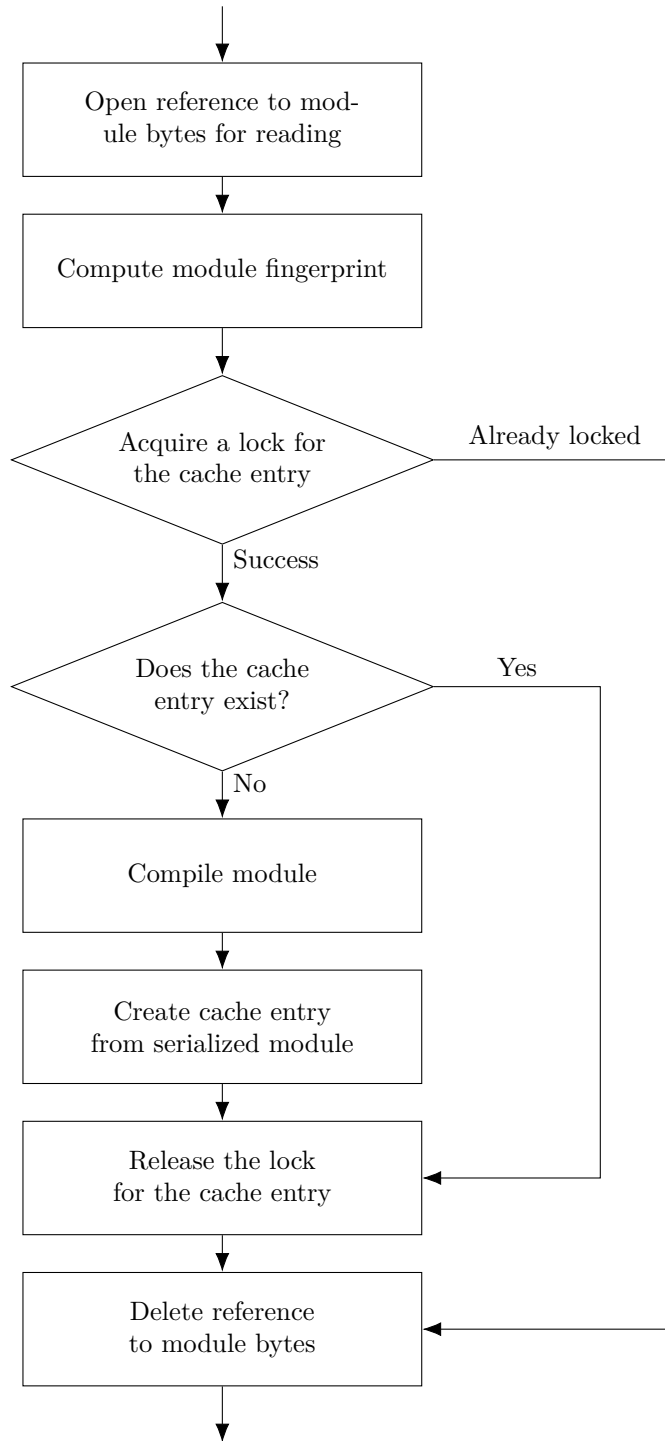


Figure 5.14: Cache design: Compiler process control flow

The compiler processes execute JavaScript code and serialization logic written in C++. The actual compilation procedures are an existing part of V8.

Most operating systems allow protecting shared memory segments from unintended write access. It appears that such measures allow controlling read and write access to the shared code cache sufficiently to prevent malicious code injections, for example, by only giving write access to the compiler processes, and not to client processes.

5.3.4 Evaluation

To evaluate our design and implementation, we consider two cases:

Cache miss: When a client process fails to locate a compiled WebAssembly module in the shared cache, it not only needs to compile the module itself, but also suffers from two additional performance impairments. First, the client process needs to copy the WebAssembly module bytes to a shared memory segment, and notify the server process about the cache miss. Depending on the module size, this can cause a short delay before compilation begins. Second, while the client process compiles the module itself, the server process will spawn a compiler process, which also compiles the module, effectively increasing the system load, and potentially increasing compilation times.

Cache hit: Upon successfully locating a compiled WebAssembly module in the shared cache, the client process benefits from two performance aspects. First, it does not need to compile the module, which, on average, improves the time until the module is available, and likely reduces CPU load and memory footprint (see Section 5.2). According to the model proposed by Patros et al. [49], this also reduces performance

interference on co-located cloud tenants. Second, if not forbidden by the process's V8 configuration, the obtained compiled code is already optimized, which would not be the case with V8's default *tiering-up* behavior, or when using Liftoff. As we have seen in Section 4.2, this can lead to improved execution times.

In the following experiments, we want to compare these two cases to the application behavior without our shared code cache.

5.3.4.1 Evaluation using PolyBench/C

While we already know the impact of deserialization as compared to compilation based on Section 5.2, we used PolyBench/C [51], which we already used in Section 4.2 and Section 4.3.3, to create a set of artificial Node.js applications to evaluate the performance impact of the shared compiled code cache. We measured the real elapsed time it takes for each application to compile and then execute its associated PolyBench/C benchmark. For this experiment, we use the default V8 configuration, which enables both Liftoff and TurboFan, and uses tiering-up (see Section 4.1), and ran each application 45 times.

Figure 5.15 shows the mean execution times for cache misses and cache hits, with error bars corresponding to the standard deviation. Since execution times between benchmarks vary tremendously, all execution times were divided by the same measurement taken without a cache in place. Similarly, the ratio between execution time and compilation time varies greatly, therefore, we do not display compilation and execution times in a stacked manner.

As expected, we see a performance regression for cache misses. It is worth noting that the benchmarks with the largest (by percentage) performance regressions such

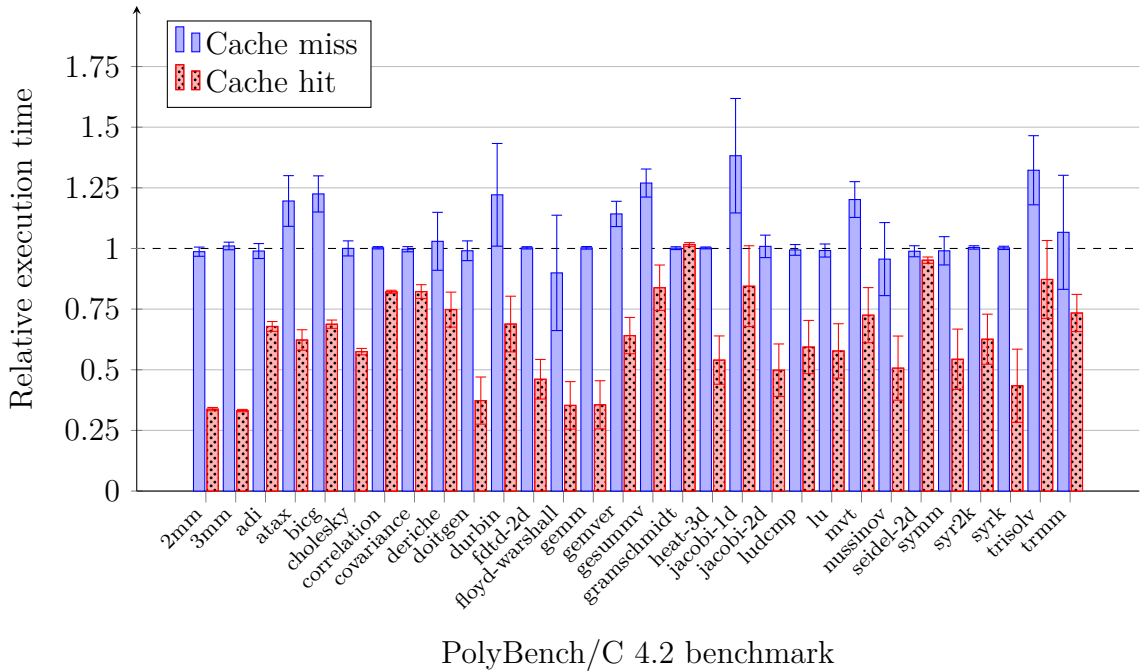


Figure 5.15: Shared cache performance impact on PolyBench/C benchmarks

as `jacobi-1d` are particularly short-running, which means that the delay caused by copying the module bytes to shared memory has a larger (by percentage) impact on the total elapsed time.

We also observe performance improvements for almost all benchmarks when a cache entry is found. The average speedup is 1.8, and the maximum speedup is 3.0. The geometric mean of the speedup across all benchmarks is 1.7. We expect that different Node.js applications would see vastly different performance benefits, depending on the WebAssembly modules in use.

5.3.4.2 Evaluation using RustPython

While the previous experiment showed execution time improvements by using the shared code cache, it uses unusually small WebAssembly modules, which do not result in realistic compile time improvements.

In this experiment, we use RustPython [3], which is a Python 3 interpreter written in Rust [35], and compiled to WebAssembly. The WebAssembly module has a size of 22 MiB and uses the WebAssembly System Interface (see Section 3.3) instead of system calls.

The Node.js application whose performance we intend to measure works as follows. First, it loads the WebAssembly module and compiles it. Immediately afterwards, it invokes the Python interpreter through a newly created WebAssembly instance (see Section 2.2) of the compiled module to execute the following line of Python code:

```
sum(filter(lambda x: x % 2 == 0,
           reversed(range(int(repr(2**10))))))
```

This code snippet computes 2^{10} , converts the result to a string (`repr`), which is converted back to an integer (`int`). This result is then passed to the `range` function, which produces an iterator that allows iterating over the integers from 0 to $2^{10} - 1$. This iterator is passed to the `reversed` function, which returns a new iterator that produces the same numbers, but in reverse order. This reversed iterator is then passed to the `filter` function along with a `lambda` function expression that restricts the iterator to even numbers. Finally, this list is passed to the `sum` function.

In other words, this piece of Python code is an inefficient way to compute the sum of all even integers that are smaller than 2^{10} . It was designed to be a short, valid

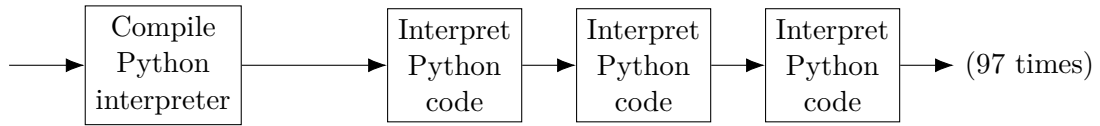


Figure 5.16: Overview of the use of the Python interpreter in the benchmarked Node.js application

Python expression that requires various computations and memory allocations and uses multiple Python data types and built-in functions.

After compiling the RustPython WebAssembly module, the Node.js application uses one hundred instances of the module to execute the above Python code one hundred times, sequentially (see Figure 5.16). Each instance executes the code once, and then terminates. Note that these instances are WebAssembly instances within the Node.js application thread, and not background threads or separate processes.

We measure the compilation time as well as the duration of each Python code execution. We run the experiment with three different configurations: first, with only Liftoff enabled, second, with only TurboFan enabled, and third, with Liftoff and TurboFan enabled and utilizing V8’s tiering-up behavior, meaning that the WebAssembly module will be compiled using Liftoff first and then again by TurboFan in the background (see Section 4.1). As in the previous experiment, we will also perform this experiment under three conditions: first, without our shared code cache, second, with our shared code cache, but with no compiled code being available in the cache (“cache miss”), and third, with our shared code cache and the compiled code being available in the cache (“cache hit”).

The combination of the three compiler configurations and the three cache-related conditions results in nine different scenarios. For each scenario, we run the Node.js

application one hundred times. As described above, each run of the application involves compiling the Python interpreter from WebAssembly to executable code for the physical hardware, as well as running the above Python code one hundred times. This total of 900 executions of the Node.js application allows us to gather performance information about 900 compilations of the WebAssembly module as well as about 90,000 executions of the Python code through WebAssembly instances of the compiled Python interpreter.

The hypothesis is twofold: First, we expect that a cache hit will reduce the delay before the first execution of Python code because it eliminates the need to compile the Python interpreter. This applies to all three compiler-related configurations (Liftoff only/TurboFan only/both via tiering-up). Second, in tiering-up mode, and based on the findings in the previous experiment, we also expect cache hits to improve the performance of Python code execution itself due to the fact that the code in the shared code cache is fully optimized, whereas V8 will optimize code just-in-time in tiering-up mode without the compiled code cache.

Therefore, we expect little difference with only Liftoff enabled since the compilation should be relatively fast and the code is unoptimized, whether it is read from the shared code cache or compiled at runtime. When only TurboFan is enabled, we expect large improvements in the initial delay, but no improvements in the execution performance. With both compilers and tiering-up enabled, we expect improvements both during startup and during execution.

Table 5.1 shows the mean compilation times for each scenario. As described above, each value is the mean of one hundred measurements.

	Liftoff	TurboFan	Tiering-up
No cache	82 ms \pm 1 ms	13130 ms \pm 196 ms	84 ms \pm 2 ms
Cache miss	99 ms \pm 3 ms	18913 ms \pm 372 ms	106 ms \pm 3 ms
Cache hit	53 ms \pm 1 ms	49 ms \pm 1 ms	51 ms \pm 1 ms

Table 5.1: Mean compilation time of the RustPython interpreter in Node.js. Errors represent one standard deviation.

Without the shared code cache, compilation using TurboFan takes significantly longer than compilation using Liftoff. Because tiering-up uses Liftoff as the first compilation stage, it causes the same small delay as Liftoff.

When a cache miss occurs, we see a substantial performance regression for all three compiler configurations. With only Liftoff enabled, this difference is small and mostly a result of the compilation function having to transfer the WebAssembly module to the shared cache server so the server can compile it for the next process. The difference is slightly larger with tiering-up enabled, and much larger (in the order of seconds) when using TurboFan. This is most likely due to the additional system load caused by the shared cache server compiling and optimizing the WebAssembly file in the background on the same system. For example, in the case of TurboFan, when a cache miss occurs, the WebAssembly module is transferred to the shared cache server, followed by the application compiling the WebAssembly module using TurboFan. At the same time, the shared cache server also begins compiling the received WebAssembly module using TurboFan. Combined, the system is under approximately twice the load than without the shared cache in place. The difference

is smaller when using tiering-up because, while the shared cache server uses TurboFan, the application process uses Liftoff first, which is much faster and uses fewer resources than TurboFan.

Most importantly, when a cache hit occurs, we see significant performance improvements in all three compilation modes: The compilation time of Liftoff was improved from 82 ms to 53 ms, and a similar improvement was observed when tiering-up is enabled. When compared to regular TurboFan compilation, the compilation time was reduced from 13130 ms to 49 ms.

As we have seen, loading the compiled code from the shared cache reduces the delay caused by compilation by 33% to 41% for Liftoff and when tiering-up is enabled, and by more than 99% when only TurboFan is enabled. In the first two cases, this means a difference of approximately 30 ms, and for TurboFan, a difference of more than 10 seconds.

When only one compiler is enabled, the performance of the generated code is virtually constant, and, therefore, the duration of each Python interpretation of the above Python code snippet is virtually constant, too.

However, when tiering-up is enabled, V8 uses TurboFan as a just-in-time compiler, and early executions of the compiled WebAssembly code are likely to use unoptimized code produced by Liftoff. When TurboFan has compiled and optimized a WebAssembly function, its next invocation will use the optimized instruction sequence instead of the instructions generated by Liftoff. Figure 5.17 shows how this behavior causes the performance of the compiled code to improve over time. The first few runs take significantly longer than later runs because the majority of functions have not been compiled by TurboFan. A few seconds later, the most relevant

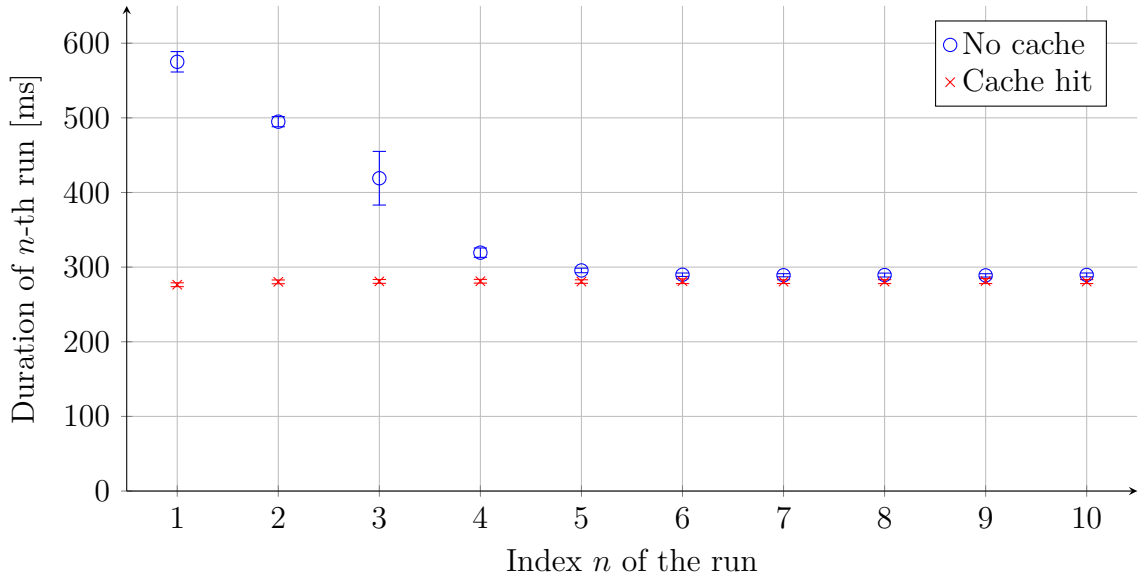


Figure 5.17: Python execution time in tiering-up compilation mode. Only the first 10 runs are shown.

functions have been compiled and optimized, and the unoptimized code produced by Liftoff has been replaced.

Also shown in Figure 5.17 are the same measurements for tiering-up compilation when the compiled code is retrieved from the cache. Because the cached code was produced by TurboFan, it is already fully optimized. Therefore, its performance is virtually constant, and each Python code execution requires approximately the same amount of time. As the graph shows, the execution times converge, and even though there is still a minor difference between the times presented in the graph, the durations of much later runs, which are not included in Figure 5.17, are virtually equal. At that point, V8 has switched from code produced by Liftoff to code produced by TurboFan entirely. However, until then, the code retrieved from the shared code cache is faster than the newly compiled code.

When only one compiler is enabled, we expect Python code execution times to be the same when no cache is in place, and when a cache hit occurs, because the compiled Python interpreter consists of exactly the same instruction sequences regardless of whether these instruction sequences were generated by the current application process or by the shared cache server. In these cases, only the compilation time is reduced, and not the Python code execution time.

Figure 5.18 shows the sum of the compilation time and the cumulative Python execution time based on the number of completed Python code executions. It shows that, regardless of the number of iterations, retrieving the fully optimized code from the cache and executing it is faster than compiling and then executing it without the shared code cache in all three compiler configurations.

When only Liftoff is enabled, the compilation delay is small (see Table 5.1), but the generated code is significantly slower than code generated by TurboFan. However, while the code produced by TurboFan provides much better performance, compilation using TurboFan causes a large initial delay due to compilation. As Figure 5.18 shows, this results in the total duration being smaller when using Liftoff than when using TurboFan up to a certain number of Python code executions, at which point the performance advantage of the code produced by TurboFan outweighs the initial delay of TurboFan compilation.

Based on Figure 5.18, tiering-up compilation appears to provide the best overall performance apart from techniques that utilize the shared code cache. While the first few Python code executions are slower than when using TurboFan because they rely on code generated by Liftoff (see Figure 5.17), the performance improves quickly until it matches that of code produced by TurboFan after a few runs.

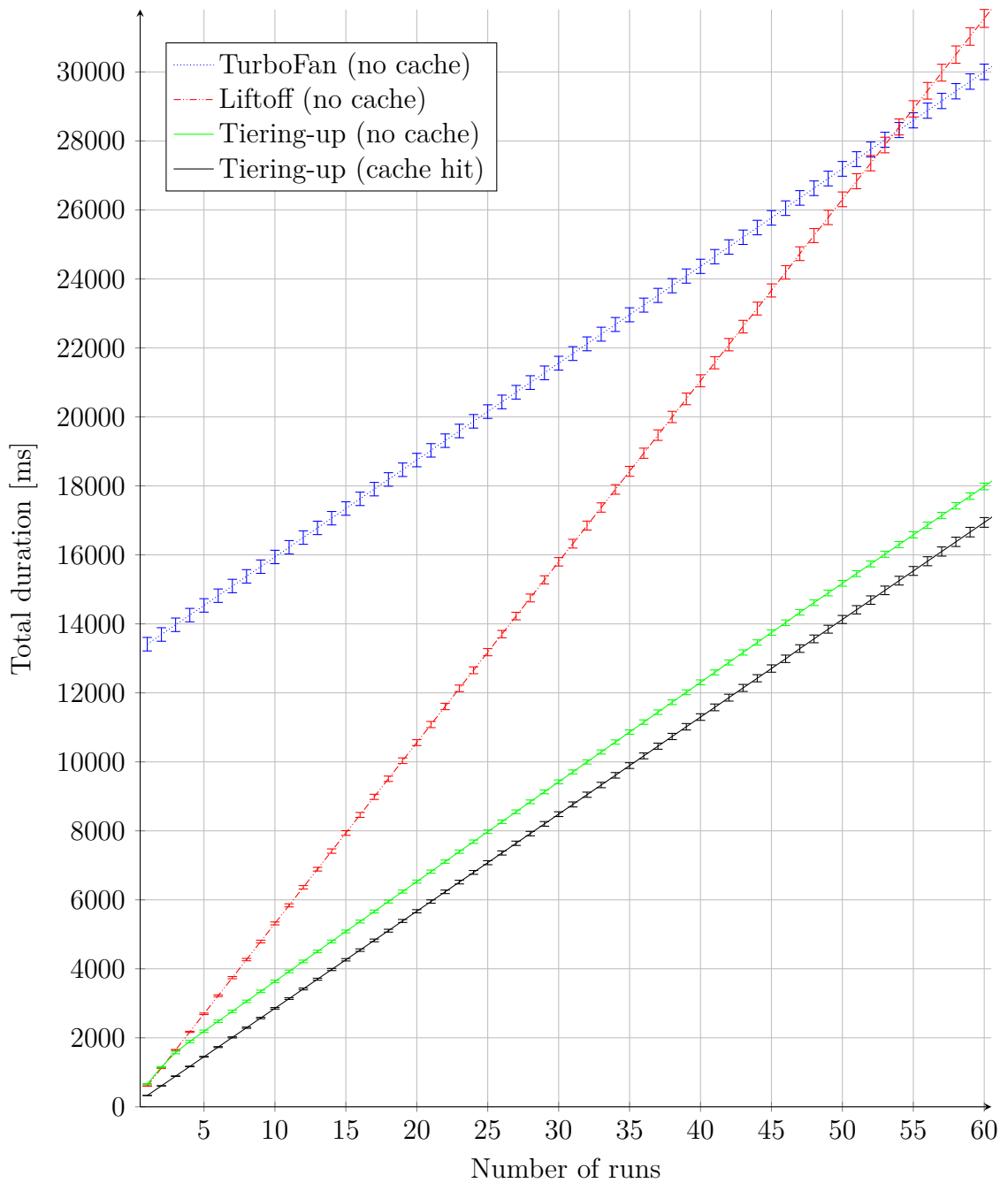


Figure 5.18: Cumulative compilation and Python execution time in selected scenarios. Error bars represent one standard deviation in each direction.

	Liftoff	TurboFan	Tiering-up
No cache	52.5 s \pm 0.5 s	41.2 s \pm 0.3 s	29.2 s \pm 0.1 s
Cache miss	52.5 s \pm 0.4 s	47.2 s \pm 0.5 s	30.3 s \pm 0.3 s
Cache hit	52.8 s \pm 0.5 s	28.2 s \pm 0.2 s	28.2 s \pm 0.2 s

Table 5.2: Total duration of WebAssembly compilation and 100 Python code executions. Errors represent one standard deviation.

However, regardless of the number of runs, tiering-up compilation and execution is still always slower than when loading the compiled and optimized code from our shared cache server. After the first few runs, the difference in the cumulative compilation and execution times remains constant. Additionally, while not part of this experiment, it is likely that tiering-up requires far more resources in terms of CPU and memory usage than restoring compiled code from the shared code cache (see Section 5.2), meaning that the visible advantage in real elapsed time is not the only benefit of code caching in this experiment.

For clarity, Figure 5.18 only shows the first 60 runs. Table 5.2 shows the mean total duration of the Node.js application in each scenario. Based on the table, code caching achieves no significant improvement with only Liftoff enabled. However, with only TurboFan enabled, the improvement is significant and exceeds ten seconds of total elapsed time. When tiering-up is enabled, code caching reduces the runtime by approximately one second when a cache hit occurs. As mentioned above, applications likely also benefit from reduced CPU and memory usage in this case (see Section 5.2).

5.4 Summary

Based on the experiments conducted in Section 5.2, we can conclude that compiled code caching can significantly and drastically reduce both compilation times and resource usage. Not only can the initial delay during the application’s startup phase be shortened, but loading optimized code from a code cache also eliminates the need for optimizing just-in-time compilation, which reduces the system load beyond the application’s startup phase.

As we have seen in Section 5.3.4.1, our shared code cache design can significantly improve the performance of computationally intensive benchmarks with respect to the default compilation procedures used by the selected Node.js version. We attribute this improvement to the fact that the code retrieved from the compiled code cache is already fully optimized and, therefore, allows benchmarks to be completed faster than when relying on just-in-time optimization without the shared code cache.

Finally, in Section 5.3.4.2, we have presented experimental results that demonstrate improved application performance as a result of our shared code cache with respect to all existing compilation strategies provided by the selected Node.js version. While tiering-up outperforms strategies that only use TurboFan and Liftoff (see Figure 5.18), using cached compiled code provides better and more consistent results due to the lack of just-in-time compilation (see Figure 5.17). This experiment only considered the elapsed real time during compilation and execution, but Section 5.2 shows that CPU load and memory usage are also reduced by caching, which means that, while tiering-up can almost match the overall application performance enabled by our compiled code cache, it requires significantly more resources.

Chapter 6

Conclusion and Future Work

This thesis has presented, discussed, and analyzed various aspects of WebAssembly with a focus on its use in Node.js applications. After an introduction to Node.js, WebAssembly, and related technologies in Chapter 2, we examined use cases, possibilities, and problems of WebAssembly integration in Node.js in Chapter 3.

As we have seen in Section 5.2, compiling WebAssembly modules at runtime can lead to a delay of multiple seconds during an application’s startup phase, and can require vast amounts of CPU time and physical memory. While the Liftoff compiler is much faster than the optimizing compiler TurboFan, its generated code is significantly slower than the code produced by TurboFan, but still much faster than interpreting WebAssembly code without compiling it first (see Section 4.2). Reasons for this performance difference were discussed in Section 4.3.

We successfully reduced WebAssembly module load times by caching compiled and optimized code for the target architecture, and observed large performance benefits for many WebAssembly modules. Finally, in Section 5.3, we extended the idea

to a scalable multi-process shared code cache, which provides an efficient way to load WebAssembly modules in Node.js applications, without having to compile and optimize each module in each process or thread. The smaller CPU demand and memory footprint can reduce interference on co-located cloud tenants, and, therefore, improve scalability [49].

In summary, the main contributions of the author’s work presented in this thesis are:

- a survey of the problems related to the use of WebAssembly modules in Node.js (see Chapter 3);
- a comparison of the compilation results produced by the WebAssembly compilers in Node.js (see Chapter 4); and
- an analysis of the performance benefits enabled by compiled code caching for WebAssembly code in Node.js applications, and the design, implementation, and experimental evaluation of a multi-process shared code cache that allows applications to utilize said benefits (see Chapter 5).

While WebAssembly is still an emerging technology, we expect growing adoption over the next few years. The performance improvements and reduced startup times presented in this thesis might allow widespread use of WebAssembly in serverless computing and other cloud configurations, without sacrificing the speed, portability, and security of WebAssembly.

6.1 Threats to Validity

We acknowledge the following threats to the validity of our results.

Confounding. While all experiments were planned, implemented, performed, and evaluated carefully, the existence of confounding factors cannot be ruled out entirely.

Differential Attrition. The set of WebAssembly modules that was used in Sections 5.1 and 5.2 originally contained a small number of modules for which compilation failed in the selected Node.js version. These modules were either invalid or relied on unsupported extensions of the WebAssembly standard and were, therefore, excluded from the experiments presented in this thesis.

Selection bias. The benchmarks and the application used to evaluate the shared code cache in Section 5.3.4.1 and Section 5.3.4.2, respectively, are not necessarily representative of real-world WebAssembly applications.

6.2 Future Work

A future direction for a shared code cache for WebAssembly code could be an extension to a disk-based cache. While the system kernel might keep frequently accessed cache entries in memory up to a certain size, large cache entries might still have to be loaded from the disk, and could negatively affect the cache performance. A balanced strategy might be to only move infrequently accessed modules from shared memory to disk when most of the available memory is in use.

While our shared cache implementation prevents duplicate compilation on the server side, it does not prevent duplicate work among client processes. The primary reason

is that client processes benefit from the shorter compilation times of tiering up, whereas the server process is focused on producing optimized code at the cost of longer compilation times. A future implementation could reduce the amount of duplicate work between processes further.

It might also be worth considering data compression for cache entries. Park et al. compressed cache entries in their JavaScript code cache implementation, and successfully reduced the size of the code cache with only minimal performance sacrifices [47]. However, as long as cache entries are stored in shared memory, decompression would require copying the decompressed data to a new memory area on each invocation, which makes it unlikely to result in large performance improvements. A disk-based cache solution could potentially benefit from compression to reduce cache entry sizes and therefore disk access times.

Lastly, Node.js [41], V8 [61], and WebAssembly [72] are rapidly changing and improving technologies. Any design or implementation of software or tools supporting these technologies must, therefore, be re-evaluated frequently, and adapted or replaced accordingly.

Bibliography

- [1] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. Serverless Computing: Current Trends and Open Problems. In Sanjay Chaudhary, Gaurav Somani, and Rajkumar Buyya, editors, *Research Advances in Cloud Computing*. Springer Singapore, Singapore, 2017. doi: 10.1007/978-981-10-5026-8_1.
- [2] Devarghya Bhattacharya, Kenneth B. Kent, Eric Aubanel, Daniel Heidinga, Peter Shipton, and Aleksandar Micic. Improving the performance of JVM startup using the shared class cache. In *2017 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, Victoria, BC, August 2017. IEEE. doi: 10.1109/PACRIM.2017.8121911.
- [3] Windel Bouwman. RustPython. <https://rustpython.github.io/>. Accessed on 2020-11-04.
- [4] Bill Budge. Code caching for WebAssembly developers, June 2019. <https://v8.dev/blog/wasm-code-caching>. Accessed on 2020-11-04.

- [5] Bytecode Alliance. WASI: The WebAssembly System Interface. <https://wasi.dev/>. Accessed on 2020-11-04.
- [6] Bytecode Alliance. Wasmtime: A small and efficient runtime for WebAssembly & WASI. <https://wasmtime.dev/>. Accessed on 2020-11-04.
- [7] Javier Cabrera Arteaga, Shrinish Donde, Jian Gu, Orestis Floros, Lucas Satabin, Benoit Baudry, and Martin Monperrus. Superoptimization of web-assembly bytecode. In *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming, '20*, page 36–40, New York, NY, USA, 2020. Association for Computing Machinery. doi: 10.1145/3397537.3397567.
- [8] Lin Clark. WebAssembly Interface Types: Interoperate with All the Things!, August 2019. <https://hacks.mozilla.org/2019/08/webassembly-interface-types/>. Accessed on 2020-11-04.
- [9] Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(2):181–196, March 1995. doi: 10.1145/201059.201061.
- [10] Ryan Dahl. Deno: A secure runtime for JavaScript and TypeScript. <https://deno.land/>. Accessed on 2020-11-04.
- [11] Michael H. Dawson, Dayal D. Dilli, Kenneth B. Kent, Panagiotis Patros, and Peter D. Shipton. Dynamically compiled artifact sharing on PaaS clouds, July 2019. Patent US 10,338,899 B2.

- [12] Craig Disselkoen, John Renner, Conrad Watt, Tal Garfinkel, Amit Levy, and Deian Stefan. Position Paper: Progressive Memory Safety for WebAssembly. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy - HASP '19*, pages 1–8, Phoenix, AZ, USA, 2019. ACM Press. doi: 10.1145/3337167.3337171.
- [13] Ecma International. *ECMAScript® 2020 Language Specification*, May 2020. <https://www.ecma-international.org/ecma-262/11.0/>. Accessed on 2020-11-04.
- [14] P. K. Gadepalli, G. Peach, L. Cherkasova, R. Aitken, and G. Parmer. Challenges and Opportunities for Efficient Serverless Computing at the Edge. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pages 261–2615, 2019. doi: 10.1109/SRDS47363.2019.00036.
- [15] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and Jf Bastien. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2017*, pages 185–200, Barcelona, Spain, 2017. ACM Press. doi: 10.1145/3140587.3062363.
- [16] Adam Hall and Umakishore Ramachandran. An execution model for serverless functions at the edge. In *Proceedings of the International Conference on Internet of Things Design and Implementation - IoTDI '19*, pages 225–236, Montreal, Quebec, Canada, 2019. ACM Press. doi: 10.1145/3302505.3310084.

- [17] Harry Halpin. The W3C Web Cryptography API: Motivation and Overview. In *Proceedings of the 23rd International Conference on World Wide Web, WWW '14 Companion*, page 959–964, New York, NY, USA, 2014. Association for Computing Machinery. doi: 10.1145/2567948.2579224.
- [18] Clemens Hammacher. Liftoff: a new baseline compiler for WebAssembly in V8, August 2018. <https://v8.dev/blog/liftoff>. Accessed on 2020-11-04.
- [19] Daniel Heidinga, Peter D. Shipton, Aleksandar Micic, Devarghya Bhattacharya, and Kenneth B. Kent. Enhancing Virtual Machine Performance Using Autonomics, March 2020. Patent US 10,606,629 B2.
- [20] David Herman, Luke Wagner, and Alon Zakai. asm.js, 2014. <http://asmjs.org/>. Accessed on 2020-11-04.
- [21] David Herrera, Hanfeng Chen, Erick Lavoie, and Laurie Hendren. Numerical computing on the web: benchmarking for the future. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages - DLS 2018*, pages 88–100, Boston, MA, USA, 2018. ACM Press. doi: 10.1145/3393673.3276968.
- [22] David Herrera, Laurie Hendren, Hangfen Chen, and Erick Lavoie. WebAssembly and JavaScript Challenge: Numerical program performance using modern browser technologies and devices. Technical Report SABLE-TR-2018-2, Sable Research Group, School of Computer Science, McGill University, Montréal, Canada, January 2018.

- [23] Mathieu Hu, Laurent Pierron, Emmanuel Vincent, and Denis Jouviet. Kaldi-web: An installation-free, on-device speech recognition system. In *INTER-SPEECH 2020 Show & Tell*, Shanghai, China, October 2020.
- [24] W. W. Hwu and P. P. Chang. Achieving High Instruction Cache Performance with an Optimizing Compiler. In *Proceedings of the 16th Annual International Symposium on Computer Architecture, ISCA '89*, page 242–251, New York, NY, USA, 1989. Association for Computing Machinery. doi: 10.1145/74925.74953.
- [25] International Organization for Standardization. Programming languages — C++. Standard ISO/IEC 14882:2017, December 2017. <https://www.iso.org/standard/68564.html>. Accessed on 2020-11-04.
- [26] International Organization for Standardization. Information technology — Programming languages — C. Standard ISO/IEC 9899:2018, June 2018. <https://www.iso.org/standard/74528.html>. Accessed on 2020-11-04.
- [27] International Organization for Standardization. Information technology — Programming languages — C#. Standard ISO/IEC 23270:2018, December 2018. <https://www.iso.org/standard/75178.html>. Accessed on 2020-11-04.
- [28] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 107–120, Renton, WA, July 2019. USENIX Association.

- [29] Faiz Khan, Vincent Foley-Bourgon, Sujay Kathrotia, and Erick Lavoie. Ostrich Benchmark Suite, June 2014. <https://github.com/Sable/Ostrich>. Accessed on 2020-11-04.
- [30] Donald E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition*. Addison-Wesley, 1981.
- [31] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004. doi: 10.1109/CGO.2004.1281665.
- [32] Daniel Lehmann, Johannes Kinder, and Michael Pradel. Everything old is new again: Binary security of webassembly. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 217–234. USENIX Association, August 2020.
- [33] Matthew C. Loring, Mark Marron, and Daan Leijen. Semantics of asynchronous javascript. In *Proceedings of the 13th ACM SIGPLAN International Symposium on Dynamic Languages, DLS 2017*, page 51–62, New York, NY, USA, 2017. Association for Computing Machinery. doi: 10.1145/3133841.3133846.
- [34] Bernd Malle, Nicola Giuliani, Peter Kieseberg, and Andreas Holzinger. The Need for Speed of AI Applications: Performance Comparison of Native vs. Browser-based Algorithm Implementations. *arXiv:1802.03707*, February 2018.

- [35] Nicholas D. Matsakis and Felix S. Klock. The rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, page 103–104, New York, NY, USA, 2014. Association for Computing Machinery. doi: 10.1145/2663171.2663188.
- [36] Hiroyuki Matsuo, Shinsuke Matsumoto, Yoshiki Higo, and Shinji Kusumoto. Madoop: Improving Browser-Based Volunteer Computing Based on Modern Web Technologies. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 634–638, Hangzhou, China, February 2019. IEEE. doi: 10.1109/SANER.2019.8668014.
- [37] Jonah Napieralla. Considering WebAssembly Containers for Edge Computing on Hardware-Constrained IoT Devices. Master’s thesis, Blekinge Institute of Technology, Karlskrona, Sweden, June 2020.
- [38] Tobias Nießen. Securing JavaScript applications with the Web Cryptography API, May 2020. <https://developer.ibm.com/technologies/javascript/articles/secure-javascript-applications-with-web-crypto-api/>. Accessed on 2020-11-04.
- [39] Tobias Nießen. synchronous-channel, February 2020. <https://www.npmjs.com/package/synchronous-channel>. Accessed on 2020-11-04.
- [40] OpenJS Foundation. Electron. <https://www.electronjs.org/>. Accessed on 2020-11-04.
- [41] OpenJS Foundation. Node.js. <https://nodejs.org/>. Accessed on 2020-11-04.

- [42] OpenJS Foundation. Node.js v14.2.0 Documentation: C++ Addons, May 2020. <https://nodejs.org/docs/v14.2.0/api/addons.html>. Accessed on 2020-11-04.
- [43] OpenJS Foundation. Node.js v14.2.0 Documentation: N-API, May 2020. <https://nodejs.org/docs/v14.2.0/api/n-api.html>. Accessed on 2020-11-04.
- [44] OpenJS Foundation. Node.js v14.2.0 Documentation: WebAssembly System Interface (WASI), May 2020. <https://nodejs.org/docs/v14.2.0/api/wasi.html>. Accessed on 2020-11-04.
- [45] OpenJS Foundation. Node.js v14.2.0 Documentation: Worker Threads, May 2020. https://nodejs.org/docs/v14.2.0/api/worker_threads.html. Accessed on 2020-11-04.
- [46] Oracle Corporation. Java SE Specifications. <https://docs.oracle.com/javase/specs/>. Accessed on 2020-11-04.
- [47] Hyukwoo Park, Sungkook Kim, Jung-Geun Park, and Soo-Mook Moon. Reusing the Optimized Code for JavaScript Ahead-of-Time Compilation. *ACM Transactions on Architecture and Code Optimization*, 15(4):1–20, December 2018. doi: 10.1145/3291056.
- [48] Panagiotis Patros, Dayal Dilli, Kenneth B. Kent, and Michael Dawson. Dynamically Compiled Artifact Sharing for Clouds. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 290–300, Honolulu, HI, USA, September 2017. IEEE. doi: 10.1109/CLUSTER.2017.9.

- [49] Panagiotis Patros, Stephen A. MacKay, Kenneth B. Kent, and Michael Dawson. Investigating Resource Interference and Scaling on Multitenant PaaS Clouds. In *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering, CASCON '16*, pages 166–177, USA, 2016. IBM Corp. doi: 10.5555/3049877.3049894.
- [50] Maria Patrou, Kenneth B. Kent, and Michael Dawson. Scaling Parallelism Under CPU - Intensive Loads in Node.js. In *27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 205–210, February 2019. doi: 10.1109/EMPDP.2019.8671573.
- [51] Louis-Noel Pouchet and Tomofumi Yuki. PolyBench/C, 2016. <https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/>. Accessed on 2020-11-04.
- [52] Jonathan Protzenko, Benjamin Beurdouche, Denis Merigoux, and Karthikeyan Bhargavan. Formally Verified Cryptographic Web Applications in WebAssembly. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1256–1274, San Francisco, CA, USA, May 2019. IEEE. doi: 10.1109/SP.2019.00064.
- [53] Micha Reiser and Luc Bläser. Accelerate JavaScript applications by cross-compiling to WebAssembly. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages - VMIL 2017*, pages 10–17, Vancouver, BC, Canada, 2017. ACM Press. doi: 10.1145/3141871.3141873.

- [54] Andreas Rossberg. Reference Types Proposal for WebAssembly. <https://github.com/WebAssembly/reference-types>. Accessed on 2020-11-04.
- [55] Alan Jay Smith. Cache Memories. *ACM Comput. Surv.*, 14(3):473–530, September 1982. doi: 10.1145/356887.356892.
- [56] James E. Smith. A study of branch prediction strategies. In *25 Years of the International Symposia on Computer Architecture (Selected Papers)*, ISCA '98, page 202–215, New York, NY, USA, 1998. Association for Computing Machinery. doi: 10.1145/285930.285980.
- [57] E. Stark, M. Hamburg, and D. Boneh. Symmetric Cryptography in JavaScript. In *2009 Annual Computer Security Applications Conference*, pages 373–381, 2009. doi: 10.1109/ACSAC.2009.42.
- [58] The LLVM Project. The LLVM Compiler Infrastructure. <https://llvm.org/>. Accessed on 2020-11-04.
- [59] Seth Thompson. Experimental support for WebAssembly in V8, March 2016. <https://v8.dev/blog/webassembly-experimental>. Accessed on 2020-11-04.
- [60] Stefan Tilkov and Steve Vinoski. Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing*, 14(6):80–83, 2010. doi: 10.1109/MIC.2010.145.
- [61] V8 Team. V8 JavaScript engine. <https://v8.dev/>. Accessed on 2020-11-04.
- [62] V8 Team. Launching Ignition and TurboFan, May 2017. <https://v8.dev/blog/launching-ignition-and-turbofan>. Accessed on 2020-11-04.

- [63] Kenton Taylor Varda, Zachary Aaron Bloom, Marek Przemyslaw Majkowski, Ingvar Stepanyan, Kyle Kloepper, Dane Orion Knecht, John Graham-Cumming, and Dani Grant. Cloud computing platform that executes third-party code in a distributed cloud computing network, June 2019. Patent US 10,331,462 B1.
- [64] Nischay Venkatram. Benchmarking AssemblyScript for Faster Web Applications. Master’s thesis, Iowa State University, Ames, Iowa, 2020.
- [65] W3C. *Indexed Database API 2.0*, January 2018. <https://www.w3.org/TR/IndexedDB-2/>. Accessed on 2020-11-04.
- [66] W3C. *WebAssembly Core Specification*, December 2019. <https://www.w3.org/TR/wasm-js-api-1/>. Accessed on 2020-11-04.
- [67] W3C. *WebAssembly JavaScript Interface*, December 2019. <https://www.w3.org/TR/wasm-core-1/>. Accessed on 2020-11-04.
- [68] W3C. *WebAssembly Web API*, December 2019. <https://www.w3.org/TR/wasm-web-api-1/>. Accessed on 2020-11-04.
- [69] Luke Wagner and Francis McCabe. Interface Types Proposal for WebAssembly. <https://github.com/WebAssembly/interface-types>. Accessed on 2020-11-04.
- [70] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. CT-wasm: Type-driven secure cryptography for the web ecosystem. *Proceedings of the ACM on Programming Languages*, 3(POPL), January 2019. doi: 10.1145/3290390.

- [71] WebAssembly Community Group. WABT: The WebAssembly Binary Toolkit. <https://github.com/WebAssembly/wabt>. Accessed on 2020-11-04.
- [72] WebAssembly Community Group. WebAssembly. <https://webassembly.org/>. Accessed on 2020-11-04.
- [73] WebAssembly Community Group. WebAssembly proposals. <https://github.com/WebAssembly/proposals>. Accessed on 2020-11-04.
- [74] E. Wen and G. Weber. Wasmachine: Bring iot up to speed with a web-assembly os. In *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 1–4, 2020. doi: 10.1109/PerComWorkshops48775.2020.9156135.
- [75] Allen Wirfs-Brock and Brendan Eich. JavaScript: The First 20 Years. *Proc. ACM Program. Lang.*, 4(HOPL), June 2020. doi: 10.1145/3386327.
- [76] Alon Zakai. Emscripten. <https://emscripten.org/>. Accessed on 2020-11-04.
- [77] Alon Zakai. Why WebAssembly is Faster Than asm.js, March 2017. <https://hacks.mozilla.org/2017/03/why-webassembly-is-faster-than-asm-js/>. Accessed on 2020-11-04.
- [78] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. Narrowing the gap between serverless and its state with storage functions. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC '19*, page 1–12, New York, NY, USA, 2019. Association for Computing Machinery. doi: 10.1145/3357223.3362723.

- [79] Aleksandra Zhuravleva. Progressive Web Camera Application using OpenCV WebAssembly module. Master's thesis, Aalto University, 2020. <http://urn.fi/URN:NBN:fi:aalto-202008234916>.

Vita

Candidate's full name: Tobias Nießen

University attended:

- Bachelor of Science in Computer Science, Gottfried Wilhelm Leibniz Universität Hannover, 2018.

Publications:

- Tobias Nießen, Michael Dawson, Panos Patros, Kenneth B. Kent. Fast Startup for WebAssembly Code. In *The IP.com Journal*, September 8, 2020. IPCOM000263535D.
- Tobias Nießen, Michael Dawson, Panos Patros, Kenneth B. Kent. Insights into WebAssembly: Compilation Performance and Shared Code Caching in Node.js. In *Proceedings of the 30th Annual International Conference on Computer Science and Software Engineering (CASCON x EVOKE 2020)*, Canada, November 10–13, 2020. doi: 10.5555/3432601.3432623

Conference Presentations:

- Tobias Nießen, Michael Dawson, Panos Patros, Kenneth B. Kent. Insights into WebAssembly: Compilation Performance and Shared Code Caching in Node.js. *30th Annual International Conference on Computer Science and Software Engineering (CASCON x EVOKE 2020)*, Canada, November 13, 2020.