JOHANNES KEPLER
UNIVERSITY LINZ

Author
Florian Huemer

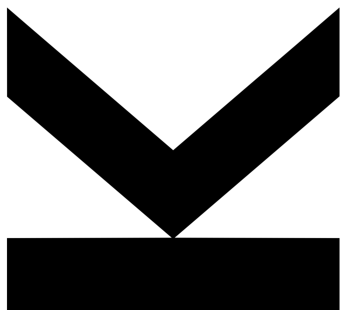
Submitted at the
Institute for System Software

Supervisor
Prof. Dr.Dr.h.c.
Hanspeter Mössenböck

Co-Supervisor
Dr. **Christian Wirth**

January 2023

Full WASM Support for GraalVM Node.js Applications



Master Thesis

to obtain the academic degree of

Diplom-Ingenieur

in the Master's Program

Computer Science

JOHANNES KEPLER
UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Austria
www.jku.at
DVR 0093696

JOHANNES KEPLER
UNIVERSITY LINZ

Kurzfassung

Das Web ist eine Platform stetiger Weiterentwicklung, die immer anspruchsvollere Anwendungen wie Videobearbeitungssoftware und Game Engines hervorbringt. Um den Anforderungen dieser hoch-performanten Anwendungen gerecht zu werden, wurden die bereits bestehenden Web-Technologien HTML, CSS und JavaScript, um WebAssembly, eine Ausführungsplattform für performance-kritischen Code im Web, erweitert. Trotz des Namesbestandteils Web, ist WebAssembly nicht auf Webbrowser limitiert. Es kann ebenso in anderen Bereichen wie IoT oder Serverless Computing eingesetzt werden. Weil WebAssembly für die Einbettung in Host-Umgebungen konzipiert wurde, werden effiziente Virtuelle Maschinen zur Ausführung benötigt. Da die GraalVM in Kombination mit dem Truffle Sprach-Implementierungs-Framework eine solide Basis für die Implementierung neuer Laufzeitumgebungen darstellt, entstand GraalWasm als eine WebAssembly Laufzeitumgebung basierend auf der GraalVM. Obwohl GraalWasm bereits den Großteil der WebAssembly Spezifikation sowie das WebAssembly System Interface unterstützt, gibt es in den Bereichen Leistung, Unterstützung von Werkzeugen und neuen Sprachfunktionen noch Aufholbedarf. Daher verbessert diese Masterarbeit GraalWasm durch die Einführung neuer Laufzeit- und Speichermodelle in den Bereichen Interpreter-Geschwindigkeit, Peak Performance und Speicheraufwand. Es erweitert die Unterstützung von Werkzeugen durch die Anpassung des existierenden Debuggers, und es bringt GraalWasm durch die Implementierung neuer Sprach-Erweiterungen näher an die Version 2.0 des WebAssembly-Standards heran.

JOHANNES KEPLER
UNIVERSITY LINZ

Abstract

The web is a platform of constant evolution, yielding more and more sophisticated applications such as video editing software or game engines. To meet this demand for high-performance applications on the web, the existing web technologies HTML, CSS, and JavaScript were extended by WebAssembly, a compilation target for the web. Contrary to its name, WebAssembly does not require a web browser, which allowed it to quickly emerge in other computing areas like IoT or serverless computing. As WebAssembly is designed to be embedded into a host environment, efficient virtual machines are needed for its execution. Since the GraalVM combined with the Truffle language implementation framework provides a solid basis for new language runtimes, GraalWasm emerged as a WebAssembly runtime based on the GraalVM. Although GraalWasm already supports the core of the WebAssembly specification as well as the WebAssembly System Interface, it still lacks behind in the areas of performance, tooling, and language features. Therefore, this thesis enhances GraalWasm by introducing new runtime and memory access models to improve interpreter speed, peak performance, and memory overhead, extends tooling support by adapting the existing debugger, and advances GraalWasm towards version 2.0 of the WebAssembly standard by implementing new language proposals.

JOHANNES KEPLER
UNIVERSITY LINZ

Contents

1 Introduction 1

2 Background 3
- 2.1 WebAssembly . . . 3
  - 2.1.1 WebAssembly Code and Data Representation . . . 5
  - 2.1.2 WebAssembly Distribution Formats . . . 8
  - 2.1.3 WebAssembly Feature Standardization Process . . . 9
- 2.2 JavaScript . . . 9
  - 2.2.1 WebAssembly JavaScript Interface . . . 10
  - 2.2.2 Node.js . . . 11
- 2.3 GraalVM . . . 12
  - 2.3.1 Truffle Framework . . . 13
  - 2.3.2 GraalWasm . . . 13
  - 2.3.3 Graal.js . . . 17
  - 2.3.4 Truffle Polyglot API . . . 18

3 Full WASM Support for GraalVM 20
- 3.1 Performance . . . 20
- 3.2 Tooling . . . 21
- 3.3 Language Features . . . 21
- 3.4 Running Example: Sieve . . . 22

4 Performance 23
- 4.1 Metrics . . . 23
- 4.2 Flat Bytecode Interpreter Model . . . 25
  - 4.2.1 Extra Data Format . . . 28
  - 4.2.2 On-Stack Replacement . . . 30
- 4.3 Static Frame API . . . 30
- 4.4 Memory Access . . . 32
- 4.5 Removal of Redundant Data in the Bytecode . . . 33
- 4.6 Evaluation . . . 34
  - 4.6.1 Benchmarks . . . 34
  - 4.6.2 Setup . . . 35
  - 4.6.3 Interpreter Performance Results . . . 36

4.6.4 Peak Performance Results . . . 38
4.6.5 Memory Overhead Results . . . 40

5 Tooling **42**
5.1 Debugger Adaptation . . . 44
5.2 Debugger Testing . . . 45

6 Language Features **46**
6.1 Multi-value Proposal . . . 47
6.1.1 Specification . . . 47
6.1.2 WebAssembly Implementation . . . 48
6.1.3 JavaScript Implementation . . . 49
6.2 Bulk Memory Operations Proposal . . . 49
6.2.1 Specification . . . 50
6.2.2 WebAssembly Implementation . . . 50
6.3 Reference Types Proposal . . . 51
6.3.1 Specification . . . 52
6.3.2 WebAssembly Implementation . . . 54
6.3.3 JavaScript Implementation . . . 56
6.4 Memory64 Proposal . . . 57
6.4.1 Specification . . . 57
6.4.2 WebAssembly Implementation . . . 58

7 Related Work **59**
7.1 Sulong . . . 59
7.2 GraalSqueak . . . 59
7.3 TruffleWasm . . . 60

8 Future Work **61**

9 Conclusion **62**

JOHANNES KEPLER
UNIVERSITY LINZ

List of Figures

2.1 Creation of a WebAssembly instance inside JavaScript with the help of the WebAssembly JavaScript Interface. . . . 11
2.2 Components of the GraalVM. . . . 12
2.3 Class diagram of the GraalWasm data structures. . . . 17

4.1 Performance measurements in the lifetime of a JIT-compiled compilation unit on the GraalVM. . . . 24
4.2 Memory distribution of a typical WebAssembly application in GraalWasm before the implementation of the flat bytecode interpreter model. . . . 25
4.3 GraalWasm AST of the running example in Listing 3.1. . . . 26
4.4 Compact extra data format representation of a `br_if` instruction. . . . 29
4.5 Extended extra data format representation of a `br_if` instruction. . . . 29
4.6 Results of the interpreter performance benchmarks of all changes introduced by this thesis. Lower is better, as depicted by the arrows next to the benchmark names. . . . 37
4.7 Results of the peak performance benchmarks of all changes introduced by this thesis. Higher is better, as depicted by the arrows next to the benchmark names. . . . 39
4.8 Results of the memory overhead benchmarks of all changes introduced by this thesis. Lower is better, as depicted by the arrows next to the benchmark names. . . . 41

5.1 DevTools of the Microsoft Edge browser showing the debugging state of a small application. . . . 43

6.1 Extension of the compact format of a `br_if` instructions with return type indicator bits. . . . 56

Listings

2.1 WebAssembly scope and branch definitions in WebAssembly text format. 7
2.2 GraalWasm bytecode interpreter loop. 14

3.1 Pseudo code of the sieve of Eratosthenes algorithm. [16] 22

6.1 Pseudo code of the sieve of Eratosthenes algorithm including multi-value extensions. 48
6.2 Pseudo code of the sieve of Eratosthenes algorithm including reference types extensions. 53

JOHANNES KEPLER
UNIVERSITY LINZ

1 Introduction

The web is an ever-evolving platform. From static content to dynamic websites based on JavaScript frameworks towards 3D software such as game engines. The trend of putting more and more sophisticated applications onto the web resulted in a new demand for high performance. While JavaScript already meets this requirement to some degree, a new compilation target optimized for such scenarios was needed [1]. This resulted in the development of WebAssembly by the four main browser vendors [2]. Originally designed for the web, WebAssembly quickly evolved into a general-purpose compilation target, covering other computing areas including serverless computing or IoT [1].

Alongside this development, a trend towards *meeting developers where they are* evolved over the last few years. Instead of forcing developers into JavaScript when targeting the web, WebAssembly allows developers to leverage their language of choice. A special focus lies on native languages such as C, C++, or Rust. In addition, this makes it easier to port existing code bases to the web and to server-side architectures. Both of these developments gave rise to new high-performance standalone execution environments for WebAssembly applications.

One of these newly evolving execution environments is GraalWasm[1]. This fast bytecode interpreter can execute applications compiled to WebAssembly either in a standalone mode or embedded into other programs. In addition, it supports a basic version of the WebAssembly System Interface[2] (WASI) and implements the WebAssembly JavaScript Interface [3] to interact with Graal.js[3], a high-performance ECMAScript-compliant JavaScript and Node.js runtime. Both GraalWasm and Graal.js are implemented in Java based on the GraalVM [4] with the help of the Truffle language implementation framework [5].

[1] `https://www.graalvm.org/22.3/reference-manual/wasm/` (visited on 2023-01-09)
[2] `https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface` (visited on 2022-11-29)
[3] `https://www.graalvm.org/javascript` (visited on 2022-10-18)

JOHANNES KEPLER UNIVERSITY LINZ

The GraalVM is a language runtime for executing high-performance applications. It includes the Graal compiler, a highly aggressive optimizing compiler that generates fast machine code and is used to compile WebAssembly applications in the context of GraalWasm. In addition, the GraalVM provides easy interoperability across different languages, called polyglot programming, such as the interaction between GraalWasm and Graal.js. Furthermore, the GraalVM ecosystem includes Truffle, a language implementation framework. This framework allows building interpreters based on abstract syntax trees (AST) with the help of a custom domain-specific language. [4, 5]

Before this thesis project was started, GraalWasm already supported version 1.0 of the WebAssembly specification. However, the WebAssembly standard already released version 2.0 of its language specification with many features that are not yet implemented in GraalWasm. In addition, GraalWasm suffered from poor performance and poor usability alongside missing support for the tools available on the GraalVM platform.

Therefore, the goal of this thesis was to explore, improve, implement, document, and evaluate various aspects in the areas of performance, tooling, and language features in GraalWasm. This includes improving the performance of the bytecode interpreter and compiled code while minimizing the memory usage of GraalWasm. Furthermore, existing tooling support for debuggers should be adapted and improved. Lastly, new language features provided in version 2.0 of the WebAssembly specification should be implemented in GraalWasm.

The remainder of this thesis is structured in the following way:

Chapter 2 provides background information about the fundamental concepts this thesis is based upon. This includes specifics about WebAssembly and JavaScript as well as the technologies adapted as part of this thesis including GraalWasm and Graal.js. Chapter 3 gives a broader overview of the areas of improvement in this thesis and introduces a running example. Chapter 4 talks about performance improvements, the challenges involved in their implementation, and the evaluation of performance with the help of benchmarks. Chapter 5 describes the improvements made in terms of tooling support while Chapter 6 describes the implementation of new language features. Chapter 7 talks about related work including similar approaches while Chapter 8 discusses future work. Chapter 9 summarizes and concludes the thesis.

2 Background

This chapter introduces the fundamental concepts of this thesis and gives the necessary background information. It first introduces WebAssembly, the language this thesis is based upon, its purpose, its core concepts, and its distribution formats. In addition, this chapter introduces the standardization process used for introducing new features to the language. It continues by describing the JavaScript language based on the ECMAScript standard, its integration with WebAssembly, and Node.js, an established JavaScript runtime environment. It furthermore describes the GraalVM and the Truffle language implementation framework and introduces GraalWasm and Graal.js, the WebAssembly and JavaScript runtimes based on the GraalVM and Truffle. To better understand the interaction between GraalWasm and Graal.js, this chapter concludes with some background information about the GraalVM Polyglot API.

2.1 WebAssembly

More and more sophisticated applications, including 3D visualizations, games, and video editing software, are moving to the web. Based on this fact, a *World Wide Web Consortium (W3C) Community Group* formed by the four main browser vendors Google, Microsoft, Mozilla, and Apple collaboratively designed *WebAssembly*. This new compilation target for the web represents the newest addition to the three existing web standards *HTML*, *CSS*, and *JavaScript*. While JavaScript already provides a powerful programming language for the web, it is not well-equipped for the requirements of the previously mentioned highly demanding web applications. [2]

According to the official WebAssembly specification [6], the main design goals of the WebAssembly bytecode are *performance*, *safety*, *hardware-*, *language-*, and *platform-independence*. Furthermore, WebAssembly aims for a *compact* and *efficient* representation.

Performance includes parsing performance, allowing for validation and compilation to be completed in a single pass, as well as run-time performance, by promising near-native speed. WebAssembly uses a sandbox environment to guarantee memory safety and requires code to be validated before it is executed.

The WebAssembly bytecode models an abstraction of modern hardware by defining a *virtual instruction set architecture*, similar to assembly code, that allows it to be platform-independent. It requires a runtime, such as a virtual machine (VM), for the execution on a specific target platform. Since WebAssembly is not bound to a specific programming model, a wide variety of languages, including Rust, C#, C++, and Python provide tools to support WebAssembly bytecode as a compilation target [2].

While its name and original purpose imply a certain closeness to the web, the WebAssembly specification [6] does not define any web-specific features. APIs specific to the integration with JavaScript, as further described in Section 2.2.1, and the integration with web browsers are defined in separate documents. Due to this loose coupling, WebAssembly has seen usage in other areas such as cloud computing, edge computing, and IoT as described in the work by Hall and Ramachandran [7] or Wen and Weber [8]. This was further accelerated by the introduction of WASI which provides a standardized way of interacting with system resources such as files or network connections.

This allows WebAssembly applications to be executed as standalone programs or to be embedded into many different environments, such as JavaScript runtimes, as further described in the WebAssembly specification [6]. Examples of standalone execution environments are Wasmtime[1] or Wasmer[2], but also GraalWasm as further described in Section 2.3.2.

On April 19th, 2022, the W3C released the latest version of the WebAssembly specification, version 2.0, which is further described in the following sections. This version added a variety of new language extensions, including support for multiple return values, reference types, and bulk memory operations, which are added to GraalWasm as part of this thesis (see Chapter 6).

[1] `https://wasmtime.dev` (visited on 2022-11-29)
[2] `https://wasmer.io` (visited on 2022-11-29)

2.1.1 WebAssembly Code and Data Representation

WebAssembly defines *modules* as its main building blocks to represent applications in a modular way. According to Haas et al. [2], modules are the static representation of a program and define all needed components such as *functions*, *globals*, *tables*, and *memories*. Details on these components are further described in Section 2.1.1.1, Section 2.1.1.2, Section 2.1.1.3, and Section 2.1.1.4. In addition, every module can define *imports* and *exports*. Imports represent components provided by other WebAssembly modules or the embedding environment while exports represent parts of a module that should be exposed for external use.

Since modules are static program representations, the execution of an application first requires the transformation of every module into an *instance* in a process called *instantiation*. Instances are the dynamic run-time representation of an application with its imports resolved, its own memory, and its own execution state. To start an application, one of the functions exported by an instance has to be called.

2.1.1.1 Functions

Functions are used to organize the code of a module into logical pieces. They take values as input parameters and can return values, call other functions, including recursive calls, and represent a sequence of instructions that is executed based on a *stack machine model*. Here, instructions work on an implicit *operand stack* by pulling values from it or pushing values onto it. An example would be the *add* instruction that pops the two topmost values from the stack, computes their sum, and pushes the result back onto the stack. In addition, functions can declare mutable local variables that allow values to be preserved throughout several instructions. This is different from many other stack-machine-based languages and is required since WebAssembly does not allow the duplication of values on the operand stack.

The value types of parameter values, return values, operand stack values, and local variables are defined by the static type system of WebAssembly. All concrete value types are split up into three different categories. *Numeric types* represent the basic machine types including two integer types and two IEEE 754 floating point number types each available as 32-bit and 64-bit versions. All of them can be used for numeric operations and the

32-bit integer type is also used for addressing linear memory and for the indices of tables [2]. *Reference types* represent pointers to internal or external functions or objects [6]. *Vector types* represent values used by vector instructions. They are currently defined as 128-bit wide values composed of several integer values or floating point values [6]. For example, a vector could consist of four 32-bit integer values or two 64-bit integer values.

In contrast to other assembly languages, WebAssembly has a semi-structured control-flow and does not use *goto* statements to arbitrary locations. Instead, WebAssembly uses scopes defined by `block`, `loop`, and `if-else` instructions. Each scope starts at the corresponding instruction and is closed by an `end` instruction. Every `if-else` instruction is composed of two `block` scopes, one for the *then* branch and one for the *else* branch. Every scope can be targeted with unconditional `br` and conditional `br_if` instructions inside of scopes. Furthermore, WebAssembly defines a `br_table` instruction for dynamically selecting a target at run time. When targeting a `block` scope, including the *then* and the *else* branch of `if-else` instructions, the program continues at the end of the scope, while for `loop` scopes, the program continues at the beginning of the scope. All branch instructions have a *label* immediate value. These labels do not refer to concrete positions in the bytecode, but instead, reference scopes by relative nesting depth. In the case of several nested blocks, a label of 0 would indicate a jump to the end of the block that directly encloses the branch instruction. A label of 1 would indicate a jump to the end of the block that nests the block that directly encloses the branch instruction, and so on.

An example of the scope definitions and branch instructions targeting them can be seen in Listing 2.1. The example shows three scopes, one `br` instruction, and one `br_if` instruction in the WebAssembly text format. The integer values next to the `br` and `br_if` instructions indicate their labels, while text inside "(;" and ";)" represents a comment.

Lines 1 to 5 define a `loop` scope. The `br` instruction in line 3 has a target label of 0, which indicates that it targets the directly enclosing scope. In this case, this is the `loop` scope at line 1. Since this is a `loop` scope, the program continues at the top of the scope.

Lines 7 to 15 define a `block` scope, while lines 9 to 13 define an additional nested `block` scope. The `br_if` instruction in line 11 has a target label of 1. This indicates that the target scope lies one additional nesting level outside of the directly enclosing scope. In the example, this is the `block` scope at line 7. Since this is a `block` scope, the program continues at the end of the scope.

Listing 2.1: WebAssembly scope and branch definitions in WebAssembly text format.

```
1  loop (; loop_0 ;)
2      ...
3      br 0 (; continues at loop_0 ;)
4      ...
5  end
6
7  block
8      ...
9      block
10         ...
11         br_if 1 (; continues at block_1 ;)
12         ...
13     end (; block_0 ;)
14     ...
15 end (; block_1 ;)
```

2.1.1.2 Globals

Globals represent values accessible from all functions and can be exported for external use or imported from an external source. They can either be constant or mutable, define an initial value, and have a predefined value type.

2.1.1.3 Memories

Linear memory in WebAssembly represents the main storage and is defined as a large array of uninterpreted bytes. Memory is defined with a minimum and maximum size in terms of pages and can grow at run time. It uses 32-bit values for addressing and can be accessed with 8, 16, 32, or 64-bit wide load and store instructions. The initial content of memory areas can be defined with *data segments*, a sequence of bytes directly encoded in the WebAssembly bytecode. The current WebAssembly specification limits the number of memory instances to a single one per module.

2.1.1.4 Tables

Tables represent storage that can hold references to functions or other objects. They again define a minimum and maximum size in terms of elements and can grow at run time. Their main purpose is to dynamically call functions at run time with the help of `call_indirect` instructions, to emulate function pointers. Tables can also be used as general-purpose storage for external object references that can be accessed via indices represented by 32-bit values. These external object references are implemented via opaque pointers. Thus, WebAssembly applications do not know about the underlying structure or implementation details of objects, but can merely pass the objects back to the embedding environment that provided them. When tables hold references to internal functions, their initial content can be defined with *elem segments*, a sequence of function references defined in the WebAssembly bytecode.

2.1.2 WebAssembly Distribution Formats

As one of the main goals of WebAssembly is a compact and efficient code representation, WebAssembly is mainly compiled to and distributed as a bytecode format. The format defines different sections that can be used to declare all parts of a WebAssembly application. All values used in this format, such as instructions, value types, and immediate values, are represented by single bytes or a combination of several bytes and follow the grammar defined in the WebAssembly specification [6].

To further improve the compactness of the format, WebAssembly uses the Little Endian Base 128 variable-length value encoding (LEB-128) as initially defined in the *DWARF Debugging Information Format* [9]. In this format, the first bit of every byte indicates if another byte belonging to the same value follows or not.

In addition to the bytecode format, WebAssembly defines a human-readable text format equivalent to its bytecode. While this text format can be used for writing small WebAssembly applications such as tests or micro-benchmarks, it can become impractical for writing entire applications. Its main purpose, therefore, is for debugging existing WebAssembly applications and representing an easier way for reading WebAssembly code.

2.1.3 WebAssembly Feature Standardization Process

The W3C defines a six-stage standardization process[3] for proposing, evolving, and integrating new features into the WebAssembly specification. A feature starts in Stage 0 as an idea. The idea is discussed and champions, one or several people responsible for the further path of the feature, are selected. If the community group approves of the idea, the feature moves through five stages of standardization.

In each stage, the feature gets refined and certain documents and implementations must be available to move to a higher stage. Stage 2, for example, requires a specification text to be available, while Stage 3 requires the official test suite to include tests that cover the new feature. If Stage 5 is reached, the feature becomes part of the WebAssembly standard.

2.2 JavaScript

JavaScript is a general-purpose, cross-platform, object-oriented programming language developed by *Brendan Eich* at Netscape intended to run inside web browsers. It was standardized by the *Ecma Standard* [10] in 1998 as *ECMAScript* based on several originating technologies including JavaScript by Netscape and *JScript* by Microsoft. As of writing this thesis, the standard defines the 13th edition as its current version and provides yearly updates.

The language standard is currently developed by the *Technical Committee 39*[4] *(TC39)* consisting of members from different companies such as Microsoft, Google, Oracle, and many others. TC39 is responsible for all aspects related to the ECMAScript standard including the language syntax and semantics, standard libraries, as well as complementary technologies that support the language. In addition, it maintains and updates the ECMAScript specification texts and develops test suites for verifying implementing runtimes.

Due to its history as a scripting language, JavaScript is not intended to be computationally self-sufficient and requires a host environment to run in. This environment provides computational objects that can be manipulated by JavaScript and functionality that is not directly defined by the ECMAScript standard such as input and output [10].

[3] `https://github.com/WebAssembly/meetings/blob/main/process/phases.md` (visited on 2022-12-17)
[4] `https://www.ecma-international.org/technical-committees/tc39` (visited on 2022-10-13)

Established engines implementing the ECMAScript standard are *V8*[5], developed by Google, and *SpiderMonkey*[6], developed by Mozilla. Another engine implementing the standard, that is extended as part of this thesis, is Graal.js, which is further described in Section 2.3.3.

2.2.1 WebAssembly JavaScript Interface

In addition to the core WebAssembly specification, the W3C defines a JavaScript API [3] for the interaction between JavaScript and WebAssembly. This interface defines the necessary classes, objects, and functions on the JavaScript side and a set of functions on the WebAssembly side for passing and retrieving data.

One of the core concepts of this interface is the `Module` class in JavaScript for representing WebAssembly modules. This class holds the WebAssembly source code and allows the retrieval of information about the module such as imports and exports. In addition, it can be instantiated into a WebAssembly instance represented by the `Instance` class in JavaScript.

This instantiation can be performed by a call to the `WebAssembly.instantiate` function in JavaScript. The required arguments are a module and an optional object defining the imports of the module. The WebAssembly JavaScript API defines that the `WebAssembly.instantiate` function is implemented as an implicit call to the WebAssembly `module_instantiate` function that executes the actual instantiation in WebAssembly and returns the WebAssembly instance. A depiction of this instantiation process can be seen in Figure 2.1.

In addition, the API defines classes for `Memories`, `Tables`, and `Globals`. These classes represent the core WebAssembly data structures as previously defined and are created similarly to modules and instances via implicit calls. The created objects can be used to provide the imports for instances to the `WebAssembly.instantiate` function or can be manipulated via additional functions provided by the `WebAssembly` namespace in JavaScript.

[5] `https://v8.dev` (visited on 2022-10-13)
[6] `https://spidermonkey.dev` (visited on 2022-10-13)

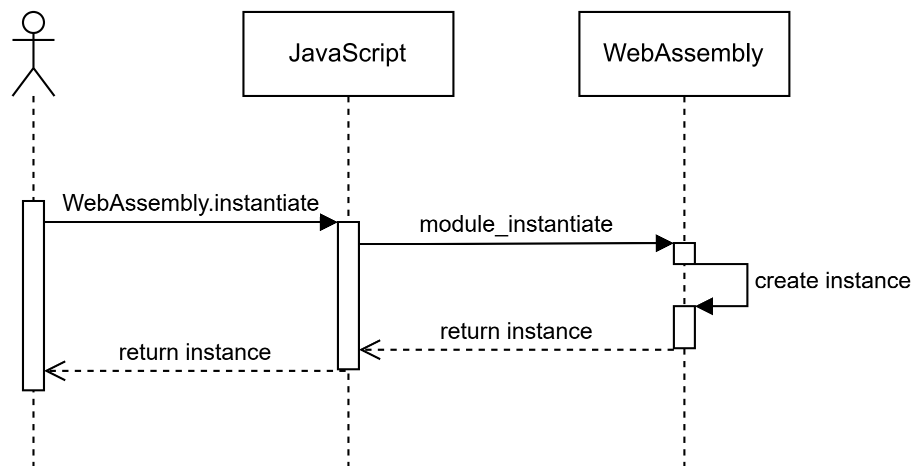


Figure 2.1: Creation of a WebAssembly instance inside JavaScript with the help of the WebAssembly JavaScript Interface.

Furthermore, the WebAssembly JavaScript API allows passing JavaScript functions as references to WebAssembly by providing them as imports to the `WebAssembly.instantiate` function. These functions can be called directly from within WebAssembly. During the instantiation of a WebAssembly module, a reference to every exported function of the WebAssembly instance is stored in an object of the `Instance` class in JavaScript. Every function is represented by a property of the object. The names of the properties are defined by the function names exported by the WebAssembly module. This allows for WebAssembly functions to be called from JavaScript.

2.2.2 Node.js

According to Liang et al. [11] Node.js is a JavaScript runtime environment based on the V8 JavaScript engine. In contrast to web browsers, Node.js allows direct access to system resources via system-level APIs. It is based on an asynchronous event-based execution model and has seen great adoption in server-side applications.

The GraalVM distributes a custom Node.js runtime[7] that replaces the V8 engine with the Graal.js JavaScript engine. This allows for Node.js applications to be executed on the GraalVM.

[7] `https://www.graalvm.org/22.3/reference-manual/js/NodeJS` (visited on 2023-01-09)

2.3 GraalVM

Optimizing language runtimes, such as VMs, for high performance often represents a barrier for newly evolving but also for established languages. While languages such as Python and PHP have seen great improvements over the last few years, they still lack behind in certain performance areas compared to highly optimized runtimes such as Oracle's Java HotSpot VM [12] or Microsoft's Common Language Runtime (dotnet VM) [13]. Unfortunately, these highly optimized runtimes often focus on a single language or a predefined set of languages compiled to a common bytecode interface. [4]

Therefore, according to Würthinger et al. [4], the GraalVM represents an alternative approach to building language runtimes by reusing the already highly optimized Java VM, based on OpenJDK or OracleJDK, as the basis for new language runtimes. It focuses on dynamically typed, imperative programming languages such as JavaScript or Python. Languages built on top of the GraalVM are mainly implemented in Java, referred to as the host language, which provides the common functionality of a VM. This no longer requires language developers to implement the core features of a VM, allowing them to focus on the required execution semantics of their guest language.

An overview of the GraalVM architecture[8] can be seen in Figure 2.2. In addition to the GraalVM compiler, it shows the Truffle language implementation framework as well as the supported guest languages of the GraalVM including JavaScript (Graal.js) and WebAssembly (GraalWasm) as described in the following sections.

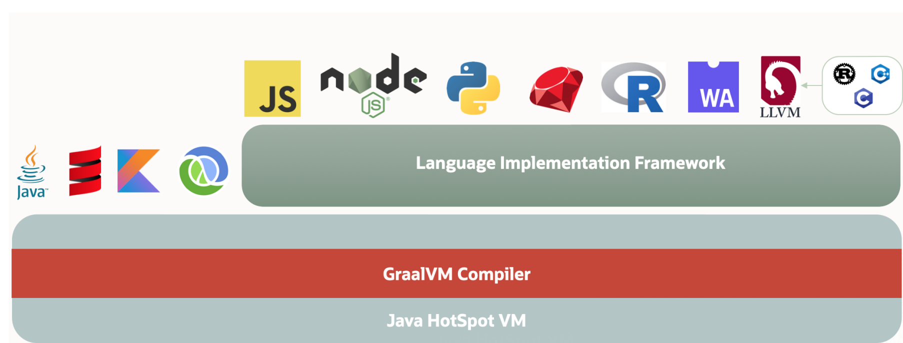


Figure 2.2: Components of the GraalVM.

[8] `https://www.graalvm.org/22.3/docs/introduction/` (visited on 2022-11-14)

2.3.1 Truffle Framework

To build new language runtimes on top of the GraalVM, Humer et al. [5] define Truffle as a framework for building AST interpreters. The framework allows defining custom node structures and specializations with the help of a custom domain-specific language based on Java's *annotation syntax*. An example of a node specialization would be an addition node that is specialized to the observed types of its operand values or an if node that is specialized based on the likelihood of taking the true branch.

The resulting interpreters are part of a multi-tier compilation system. The code of an application starts in the interpreter where profiling information is gathered. As soon as a method is considered *hot* (i.e., is executed frequently) by the interpreter, Truffle's partial evaluation (PE) engine specializes the interpreter to the observed inputs and passes it to the Graal compiler to produce optimized machine code. The compiler leverages speculative optimizations to produce specialized code based on assumptions derived from PE. In addition, deoptimization points, which transfer the execution back to the interpreter, are inserted into the code for cases where an assumption fails.

While Truffle is mainly designed for AST-based languages, Niephaus et al. [14] state that Truffle can also be used to implement bytecode interpreters. However, in this approach, the compiler requires some additional information to achieve good performance.

2.3.2 GraalWasm

The GraalVM provides a WebAssembly runtime based on the Truffle framework called GraalWasm[9]. The implementation state for this runtime is currently defined as experimental. Previous to this thesis, GraalWasm implemented version 1.0 of the WebAssembly specification and parts of WASI. GraalWasm allows the highly-performant execution of a WebAssembly engine in standalone mode or can be embedded into other languages with a special focus on JavaScript.

[9] `https://www.graalvm.org/22.3/reference-manual/wasm/` (visited on 2023-01-09)

2.3.2.1 GraalWasm Interpreter Model

As WebAssembly is a bytecode format, GraalWasm is implemented as a bytecode interpreter based on the Truffle framework. Here, the WebAssembly bytecode is represented as a Java byte array called `data`. An `offset` pointer represents the current location in the bytecode that is executed.

The interpreter itself is implemented as a loop with a large `switch` statement in it. Every valid WebAssembly instruction is represented by a single `case` label inside the `switch` and calls a method that implements the program logic of the instruction. A simplified representation of this bytecode interpreter loop can be seen in Listing 2.2.

Listing 2.2: GraalWasm bytecode interpreter loop.

```java
@BytecodeInterpreterSwitch
@ExplodeLoop(kind = ExplodeLoop.LoopExplosionKind.MERGE_EXPLODE)
public Object executeBodyFromOffset(...,VirtualFrame frame, int offset,
    int stackPointer) {
    while(offset < endOffset) {
        int opcode = readOpcode(data, offset);
        offset++;
        switch(opcode) {
            case LOCAL_GET: {
                // Read local index and advance offset pointer
                int index = readValue(data, offset);
                offset += readValueLength(data, offset);

                local_get(frame, stackPointer, index);
                stackPointer++;
                break;
            }
            case I32_EQZ: {
                i32_eqz(frame, stackPointer);
                break;
            }
            case I32_EQ: {
                i32_eq(frame, stackPointer);
                stackPointer--;
                break;
            }
            ...
```

The `@BytecodeInterpreterSwitch` and `@ExplodeLoop` annotations in lines 1 and 2 are part of the Truffle framework and define compiler hints for better code generation. The `@BytecodeInterpreterSwitch` annotation defines that a certain method, such as the root method of a bytecode interpreter, should receive some additional optimization budget. The `@ExplodeLoop` annotation defines that the loops inside of the methods should be fully unrolled. According to Leopoldseder et al. [15], this unrolling process duplicates the body of the loop in front of itself in compiled code. Fully unrolling a loop means that every iteration of the loop is unrolled. Therefore, in compiled code, the loop is represented by a linear sequence of instructions. The `kind` parameter of the `@ExplodeLoop` annotation defines the strategy for unrolling the loop and how to handle merge points resulting from control-flow instructions such as loops and conditionals. The `MERGE_EXPLODE` strategy is specifically designed for bytecode interpreters as explained by Niephaus et al. [14].

The Truffle framework requires applications to be represented by an AST. Therefore, the bytecode interpreter, represented by the `executeBodyFromOffset` method in Listing 2.2, was part of the `WasmBlockNode` class in the version of GraalWasm previous to this thesis. In addition, GraalWasm provided a `WasmRootNode` for representing functions and used the `LoopNode` provided by the Truffle framework to represent loops in the WebAssembly bytecode. Furthermore, it used a `WasmIfNode` for representing if-else constructs. The usage and purpose of these nodes inside the AST and their adaptation or removal due to changes introduced in this thesis are further described in Section 4.2.

2.3.2.2 GraalWasm Data Structures

To optimize the run-time performance of WebAssembly applications, GraalWasm defines several versions of the three main WebAssembly data structures globals, memories, and tables. They are selected based on given run-time conditions or can be activated or deactivated by the user with the help of flags provided to the runtime.

Globals are represented in two different ways, depending on their origin. If a global is created during the instantiation of a WebAssembly module, the global is stored as a primitive value or a Java object in the `GlobalRegistry` based on its value type. If the creation of the global is requested by an external source, such as the WebAssembly JavaScript Interface, a different strategy is chosen. In these cases, it is more likely that the global gets frequently passed around between JavaScript and WebAssembly. Therefore, an optimized version

represented by the `DefaultWasmGlobal` class, which directly encapsulates the value of the global, is used. This minimizes the effort for retrieving the information about the global every time it is accessed from outside WebAssembly. `DefaultWasmGlobals` are stored in a separate array in the `GlobalRegistry`. If globals defined during the instantiation process have to be passed to other languages, they are encapsulated as a reference to the registry in an `ExportedWasmGlobal`.

Furthermore, there are two different implementations of WebAssembly memory in GraalWasm represented by the `ByteArrayWasmMemory` and `UnsafeWasmMemory` classes. As the name suggests, the `ByteArrayWasmMemory` uses a Java byte array to represent the underlying memory. The `UnsafeWasmMemory` uses a direct buffer, represented by the `ByteBuffer` class[10], which is accessed via Java `Unsafe`. The `UnsafeWasmMemory` is an opt-in feature of GraalWasm and can optimize performance in certain memory-heavy use cases. It is especially useful when running WebAssembly in the context of the Node.js implementation of Graal.js. Here, certain Node.js APIs, written in native C code, can access the memory directly without having to invoke a call back to a Java method. The memory implementations are disjoint and can therefore not be used in combination. The resulting memories are stored in the `MemoryRegistry`.

WebAssembly tables are represented by the `WasmTable` class and store references to functions and external objects in a Java Object array. The resulting tables are stored in the `TableRegistry`.

To put all these parts together, GraalWasm provides the `WasmModule` class containing static information such as the symbol table, mappings of imports and exports, the source code, and information needed by the linker. The `WasmInstance` class holds dynamic run-time information such as references to globals, memories, tables, and functions. In addition, the `WasmInstance` class holds a reference to the `WasmModule`, to access static information.

A depiction of the mentioned classes and their relationships can be seen in Figure 2.3.

[10]`https://docs.oracle.com/javase/7/docs/api/java/nio/ByteBuffer.html` (visited on 2022-11-14)

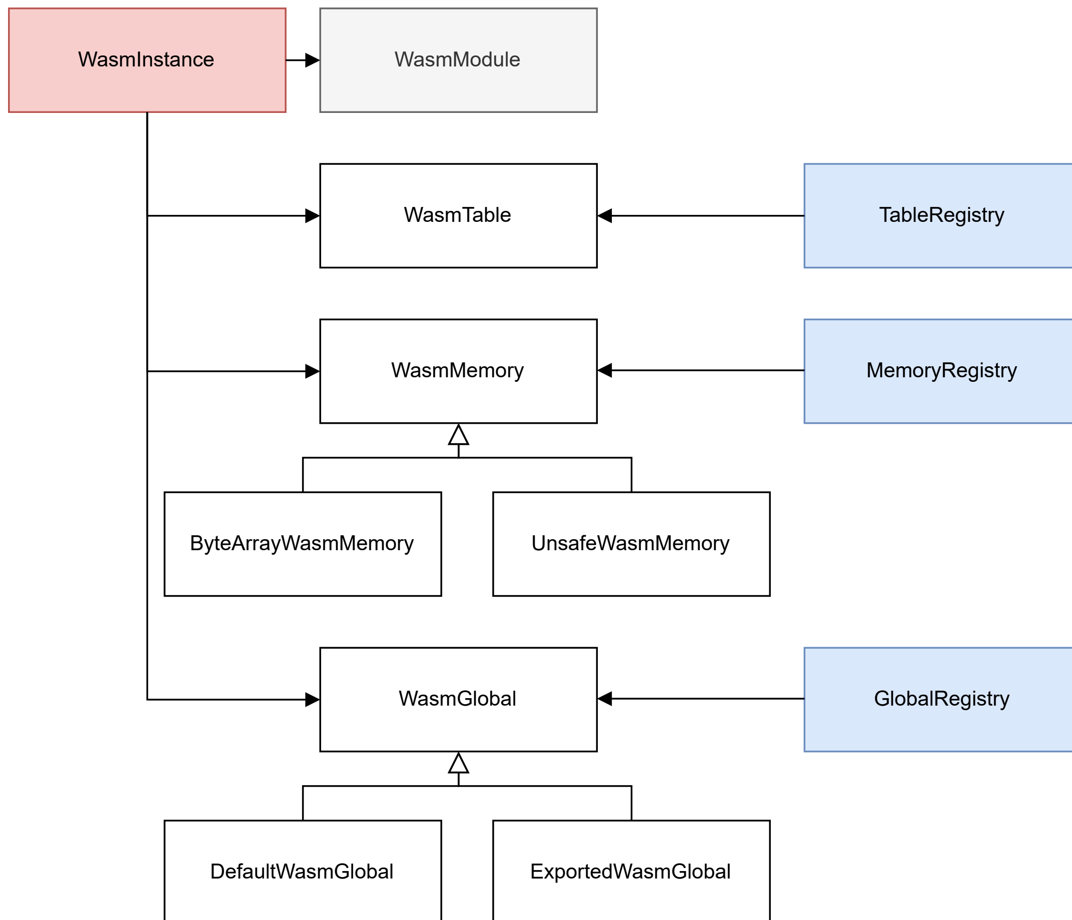


Figure 2.3: Class diagram of the GraalWasm data structures.

2.3.3 Graal.js

Graal.js[11] represents a fully standard-compliant ECMAScript implementation based on the GraalVM with support for Node.js. It is implemented as an AST interpreter based on the Truffle framework and leverages the benefits provided by the GraalVM stack, such as node specializations and common tooling. Graal.js uses the Polyglot API provided by the Truffle framework to interact with GraalWasm and implements the WebAssembly JavaScript Interface.

[11] `https://www.graalvm.org/javascript` (visited on 2022-10-18)

JOHANNES KEPLER
UNIVERSITY LINZ

Graal.js represents one of the driving factors for WebAssembly on the GraalVM. While standalone WebAssembly applications can be executed on GraalWasm, most of the time, WebAssembly is embedded in JavaScript. This is especially true when executing Node.js applications on Graal.js. Here, several existing packages use WebAssembly for use cases such as cryptography[12] or graphics processing[13]. In addition, Node.js has experimental support for WASI[14], which is implemented via GraalWasm in the Node.js version of Graal.js.

2.3.4 Truffle Polyglot API

The Truffle framework includes a set of classes designed for guest language interoperability. GraalWasm and Graal.js leverage the features provided by these classes to implement the WebAssembly JavaScript Interface.

The API is mainly built around the `TruffleObject` interface. Classes that implement the `TruffleObject` interface can choose to implement a selection of methods from a predefined set. The set contains methods for representing common object layouts such as functions, arrays, or maps.

The `WasmFunctionInstance`, representing a WebAssembly function inside GraalWasm at run time, implements the `isExecutable` and `execute` methods. This allows other languages using this object to identify that it is a function via the `isExecutable` method and allows them to execute the underlying function via the `execute` method.

Array-like objects, such as one of the GraalWasm memory representations `ByteArrayWasmMemory`, implement the `readArrayElement` and `writeArrayElement` methods in addition to some other array-related methods. This allows other languages using this object to identify it as an array and to access and modify the individual array elements.

The main class used for interop between GraalWasm and Graal.js is the `WebAssembly` class inside GraalWasm. It is represented by a map-like `TruffleObject` that provides most of the methods defined in the WebAssembly JavaScript Interface [3]. The names of the

[12] `https://www.npmjs.com/package/@bitgo/blake2b-wasm` (visited on 2023-01-16)
[13] `https://www.npmjs.com/package/canvaskit-wasm` (visited on 2023-01-16)
[14] `https://nodejs.org/api/wasi.html` (visited on 2023-01-16)

functions, such as `module_instantiate`, are the keys while their implementing methods represent the values of the map.

Graal.js can extract these functions by calling the `readMember` method on the `WebAssembly` object exported by GraalWasm. Since the exported functions are again `TruffleObjects`, Graal.js can use the `execute` methods on them to perform the needed function calls.

3 Full WASM Support for GraalVM

When thinking about usability and support in the context of software development a broad variety of aspects can be considered from both a development standpoint as well as a usage standpoint. While developers often strive for easy-to-use tooling support including debuggers and profilers together with the latest language features, users mainly benefit from high-performance and bug-free runtime environments.

Based on this assumption, we focus our work on the three key areas *performance*, *tooling*, and *language features*. Details about these areas as well as the challenges involved in implementing and improving certain aspects of the GraalWasm WebAssembly runtime are described in Chapter 4, Chapter 5, and Chapter 6.

3.1 Performance

In the context of this thesis, the area of performance mainly deals with the improvement of the interpreter performance and the peak performance of GraalWasm. Furthermore, it reduces the overall memory consumption of GraalWasm. This benefits users and developers alike. Improving performance reduces the time needed to execute an application. This allows developers and users to get the expected results of their applications faster while using less memory for the execution.

To achieve this goal, we replaced and optimized the data structures used for representing run-time data in GraalWasm, as further described in Section 4.2 and Section 4.5. Furthermore, we reduced and optimized memory accesses, as further described in Section 4.3 and Section 4.4.

The resulting changes in performance and memory consumption were quantified with the help of benchmarks. The concrete results can be found in the evaluation in Section 4.6.

3.2 Tooling

In terms of tooling, this thesis extends the existing debugger in GraalWasm to improve the debugging experience of developers. This includes the optimization of the debugger performance and implementing unit tests to extensively test the debugger.

To achieve this goal, the existing debugger was adapted to the changes introduced in Chapter 4. This allows us to further optimize the performance of the debugger, as further described in Section 5.1. In addition, a multitude of tests were added to make sure that the data depicted by the debugger is correct, as described in Section 5.2.

3.3 Language Features

This area deals with the implementation of new language features introduced in version 2.0 of the WebAssembly specification [6]. This includes the introduction of multiple return values, new instructions for dealing with large chunks of memory, and better integration of the embedding environment in WebAssembly applications with the help of reference types.

The bytecode of WebAssembly applications is mainly produced by compilers of native languages such as C or C++. These compilers leverage all language features available in the specification to produce WebAssembly applications that run with good performance. This requires WebAssembly runtimes, such as GraalWasm, to also support these language features to execute applications produced by a multitude of compilers.

Therefore, four proposals were implemented as part of this thesis to be able to execute WebAssembly applications targeting version 2.0 of the WebAssembly standard. This includes the multi-value proposal, as further described in Section 6.1, the bulk memory operations proposal, as further described in Section 6.2, and the reference types proposal, as further described in Section 6.3. Furthermore, the memory64 proposal, as further described in Section 6.4, was implemented as part of this thesis.

3.4 Running Example: Sieve

To give some guidance throughout this thesis, we introduce a small running example application. The application is concerned with determining if a given number is prime or not. It does so by leveraging an implementation of the *sieve of Eratosthenes* in WebAssembly. A pseudo code of the algorithm can be seen in Listing 3.1. We do not assume any implementation details, such as the originating source language or the used memory layout, for our example except for the control flow of the implementing function called `sieve`.

Since WebAssembly does not provide any means of interaction with the user, such as interaction with the console, we use a runtime, such as Node.js, as our embedding environment. It takes the user input from either the console or another input source, such as a file, and performs calls to the WebAssembly `sieve` function to determine if a number is prime or not. The result is then written back to the console or another form of output.

Listing 3.1: Pseudo code of the sieve of Eratosthenes algorithm. [16]

```
1   function sieve:
2       input:  integer n where n > 1.
3       output: whether n is prime or not.
4
5       let M be an array of boolean values from 2 to n, all set to true.
6
7       for i in 2, 3, 4, ... to sqrt(n):
8           if M[i] is true:
9               for j in 2 * i, 3 * i, 4 * i, ... to n:
10                  M[j] = false
11      return M[n]
```

The sieve algorithm in Listing 3.1 takes a number n greater than 1 and determines whether n is prime or not. The algorithm starts with the assumption that all numbers in the array M are prime. It then iterates over all numbers i from 2 to the square root of n and marks all multiples of i in the array as non-prime. In the end, only prime numbers have a remaining *true* value in the array. By extracting M at position n, one can determine if the given number is prime or not. [16]

4 Performance

Since WebAssembly defines performance as one of its main goals, WebAssembly runtimes, such as GraalWasm, should endeavor to optimize all performance aspects of their run time. Therefore, to further optimize the performance of GraalWasm, four changes were introduced that are described in this chapter.

To identify possible areas of improvement in GraalWasm and to verify the expected performance gains of the introduced changes, Section 4.1 describes the relevant performance metrics. Section 4.2 introduces the flat bytecode interpreter model, focusing on the reduction of memory usage, while Section 4.3 introduces the static frame API to improve interpreter speed. Furthermore, Section 4.4 describes changes to the representation of memory addresses and the resulting improvements when accessing memory while Section 4.5 describes the removal of redundant data in GraalWasm. Section 4.6 concludes by evaluating the changes against a set of micro-benchmarks.

4.1 Metrics

In the context of a just-in-time (JIT) compiler, such as the Graal compiler used by GraalWasm to JIT-compile WebAssembly code at run time, the main performance aspects are *peak performance*, *interpreter performance*, *warmup time*, and *memory overhead*. To find possible areas of improvement in GraalWasm, this thesis focused on peak performance, interpreter performance, and memory overhead.

We define *peak performance* as the throughput of an application after all hot methods have been compiled by the JIT compiler. It represents the main performance measurement for applications and is measured in operations per second in our benchmarks.

We further define *interpreter performance* as the throughput of an application running in the interpreter of a JIT-compiled runtime. At this point, none of the methods have been compiled to optimized machine code yet. Interpreter performance is relevant for the initial performance of an application and is measured in milliseconds needed to execute a single benchmark run. This metric can also be an indicator of the performance of GraalWasm when executed outside the context of the GraalVM.

Furthermore, we define *warmup time* as the time spent in the interpreter between starting the application and reaching peak performance. Warmup time should be minimized to reach peak performance as soon as possible.

A depiction of the performance aspects in relation to the lifetime of a JIT-compiled compilation unit (function) on the GraalVM can be seen in Figure 4.1.

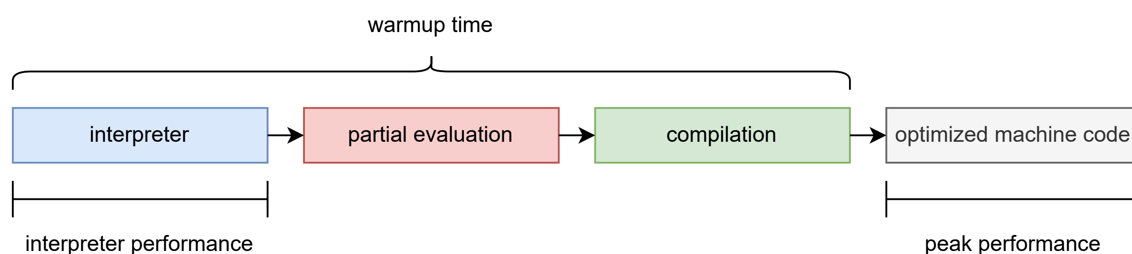


Figure 4.1: Performance measurements in the lifetime of a JIT-compiled compilation unit on the GraalVM.

In addition, we define the *memory overhead* of GraalWasm as the extra amount of memory needed in addition to the run-time data of a WebAssembly application, such as globals, memories, and tables. This includes symbol tables, profiling information, function graphs, the internal representation of the source code, and all other data needed to represent the static and dynamic state of an application. We explicitly exclude the memory needed by the GraalVM itself from our definition of memory overhead to focus on those areas that can be improved inside GraalWasm.

4.2 Flat Bytecode Interpreter Model

The first of our performance improvements, the *flat bytecode interpreter model*, is motivated by the amount of data needed to represent WebAssembly function graphs in GraalWasm. Figure 4.2 shows a summary of the distribution of the memory overhead of a typical WebAssembly application based on results from our benchmarks. The data was extracted with the help of the Java Object Layout[1] tool and was categorized into five main categories that represented the largest junks of memory.

It can be seen that the function graphs, each represented by an AST, account for nearly half of the overall memory overhead. Another 18% is needed by the linker for representing *link actions*. These functions are needed to instantiate modules into instances. The internal representation of source files and source code accounts for 13% each. Everything else, such as entries in the symbol table or the memory overhead produced by the global registry, memory registry, and table registry, are categorized into *other* and are not considered in the flat bytecode interpreter model.

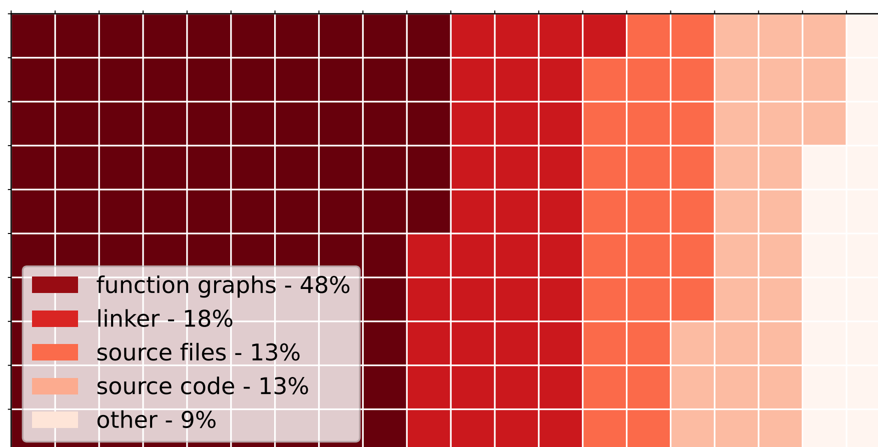


Figure 4.2: Memory distribution of a typical WebAssembly application in GraalWasm before the implementation of the flat bytecode interpreter model.

[1]`https://github.com/openjdk/jol` (visited on 2023-01-04)

To put the memory usage of function graphs, which account for most of the memory overhead, into perspective, Figure 4.3 depicts the AST of our running example introduced in Chapter 3.

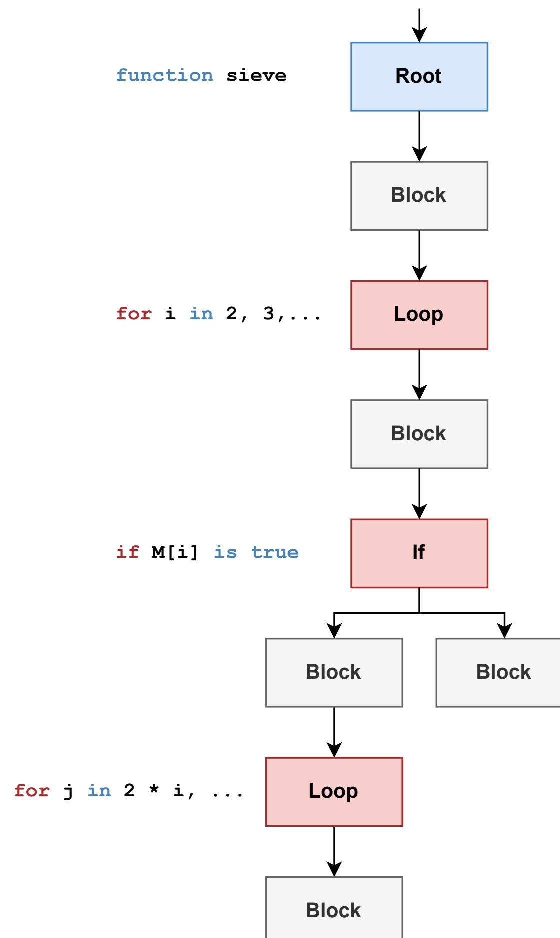


Figure 4.3: GraalWasm AST of the running example in Listing 3.1.

The *root node* of the AST represents the entry point into the function. Before executing the WebAssmbly code, it extracts the arguments provided by the caller and puts them onto the operand stack. After the code execution is finished, it pops the result value from the operand stack and returns it as a Java object.

Block scopes are represented by *block nodes*. In addition to defining a branch target, they execute the code sequence contained in the block scope in a bytecode interpreter loop.

Control structures, such as `loops` and `ifs`, are represented by *loop nodes* and *if nodes*. The body of every `loop` as well as both branches of an `if` are again represented by separate block nodes.

Branch instructions targeting blocks are implemented as a cascade of returns. This means that a branch instruction executed inside a block node exits the current interpreter loop and returns the target label of the branch instruction, an integer value representing the relative nesting depth. The parent block node takes this target label and checks if it is zero. If the target label is zero, the cascade stops and the current block continues execution. Otherwise, it decreases the target label by one and propagates it to its parent block node.

Every block node stores its start offset and end offset in the bytecode, its initial operand stack pointer, and some additional fields and arrays for storing profiling information. The bytecode itself only exists once and is not copied to every node.

According to Prokopec[2], this approach was originally chosen to get the best of both interpreter approaches, an AST for control flow and a bytecode interpreter loop for executing instructions similar to Figure 4.3. While this approach already produces good results in terms of memory overhead, we can show that the bytecode interpreter can be improved by flattening it, which reduces the memory overhead even further.

In our approach, the flat bytecode interpreter model, control flow is no longer represented by individual nodes. Instead, control flow is performed inside the bytecode interpreter loop by setting the offset pointer to predefined target locations in the bytecode when performing a jump. This approach is similar to control flow in the bytecode of other languages such as the Java Bytecode used by the Java Virtual Machine[3].

The information needed for performing these jumps as well as the profiling information needed for conditional jumps is stored in an additional integer array. This array is from here on referred to as *extra data array* while the content of the array is referred to as *extra data*. Examples of instructions needing extra data are `if`, `br`, and `br_if`, which store a branch target, represented by a relative offset to a fixed location in the bytecode, in addition to some profiling information.

[2] `https://medium.com/graalvm/announcing-graalwasm-a-webassembly-engine-in-graalvm-25cd040\0a7f2` (visited on 2022-11-21)
[3] `https://docs.oracle.com/javase/specs/jvms/se19/html/` (visited on 2023-01-24)

Since the WebAssembly bytecode uses relative targets based on the nesting depth of an instruction, this information first has to be translated into relative offsets to fixed locations in the bytecode by the GraalWasm bytecode parser. The resulting data is stored in a format specifically optimized for memory consumption as explained in the following section.

The extra data format and the optimized control-flow implementation allow for every WebAssembly function to be represented by two nodes. A root node and a block node which is from now on referred to as a *function node*. The function node stores the start offset and end offset for the entire function and an extra data array. Since all the profiling information is now part of the extra data array, no additional fields or arrays are needed any longer.

This results in an up to 90% reduction of the memory needed for function graphs and a reduction of up to 42% of the overall memory overhead. In addition, this change improves the performance of the interpreter by up to 23%. While previously, the execution of every node represented an individual method call, the number of method calls in the interpreter needed for control flow is reduced to a single one per WebAssembly function in the flat bytecode interpreter model. Furthermore, due to the reduction of nodes in the AST, the compiler can apply more optimizations to it, which results in better peak performance. The resulting benchmark numbers can be seen in Section 4.6.

4.2.1 Extra Data Format

To minimize the overhead introduced by the extra data array, the encoding used for storing the information needed by certain instructions is available in two formats, a *compact* and an *extended* format. The compact format is limited to a fixed set of value ranges that is not exceeded by a typical WebAssembly application. If larger values are needed, the extended format provides enough space to support the full range of values defined by the WebAssembly specification [6].

Extra data is needed by every single `if`, `else`, `br`, `br_if`, `br_table`, `call_indirect`, and `call` instruction. Each of the instructions can either use the compact or the extended format. To get a better understanding of the compact and the extended format, Figures 4.4 and 4.5 show a comparison of the extra data needed by an individual `br_if` instruction.

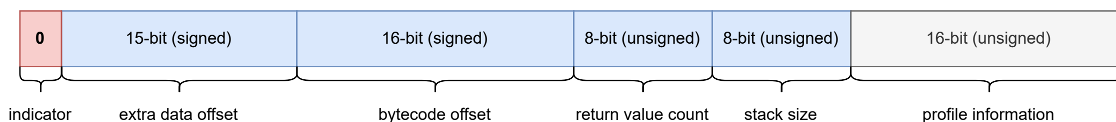


Figure 4.4: Compact extra data format representation of a `br_if` instruction.

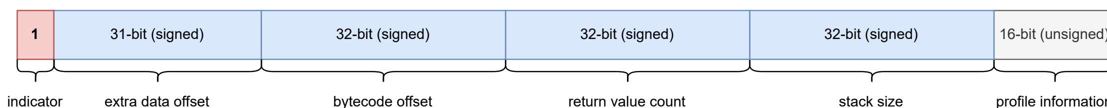


Figure 4.5: Extended extra data format representation of a `br_if` instruction.

Both encodings use the first bit of an entry to indicate if the entry is in the compact format, indicated by a zero, or the extended format, indicated by a one. The second entry represents the relative target offset in the extra data array. This is needed since jumps do not only change the position in the bytecode, but also the position in the extra data array. The third entry is the relative target offset in the bytecode. Since `block` scopes and `if-else` instructions can have return values, the fourth value indicates the number of return values of the target scope. Before the introduction of the multi-value proposal, as further described in Section 6.1, the number of result values was limited to a single result per scope. With this in mind, one bit would have sufficed for the number of return values in both cases. Since the implementation of the multi-value proposal was already planned at this point, we decided to reserve 8 bits in the compact format to allow for several return values. The fifth value indicates the stack size of the target scope while the sixth value is reserved for profiling information.

Both the result value count and the stack size are needed for *unwinding* the stack when a jump is performed as described by Haas et al. [2]. Since the currently executed scope might use more values on the operand stack than the target scope of a branch instruction, the unwind operation first copies the result values from the top of the stack to the stack location indicated by the stack size value. This results in a new top-of-stack. All values beyond this new top-of-stack are dropped and the execution continues at the target location of the branch in the bytecode.

4.2.2 On-Stack Replacement

According to Fink and Qian [17], *on-stack replacement* (OSR) is a technique used by dynamic compilers to reduce warmup time. OSR works by replacing hot methods with their compiled version while a method is still executed. This is especially useful for methods with long-running loops. Without OSR, the optimized compiled version would only be used when the respective method is called the next time and would not affect already running methods. Since functions are the smallest optimization unit in most compilers, loops have to be extracted into separate functions to support OSR.

This operation is already supported in GraalVM with the help of the `LoopNode`. Since the flat bytecode interpreter model no longer uses `LoopNodes` to represent loops, a different approach introduced by Mosaner et al. [18] has to be applied. In this approach, loops are reconstructed from the control flow of an application. To support this approach, the flat bytecode interpreter model explicitly tries to invoke OSR compilation when encountering a `loop` instruction. The GraalVM decides to perform OSR based on a set of heuristics including the number of loop iterations that are reported by GraalWasm.

4.3 Static Frame API

Memory access can have a great impact on the performance of interpreters. Therefore, our approach tries to minimize the number of array reads and writes in the GraalWasm interpreter. While the extra data array was designed with this goal in mind, the data structure storing the operand stack in GraalWasm, called the *Frame*, was further optimized with the introduction of the *static frame API*.

According to the Truffle API documentation, the Frame[4] represents the local variables of a guest language. In the case of GraalWasm these are the local variables and the operand stack. Since the GraalVM focuses mainly on dynamically typed languages, the Frame was designed with this information in mind. This primarily manifests itself by the fact that every access to a location, called a *slot*, on the Frame performs a type check. For example, the `getInt(int slot)` method of the Frame first checks if the slot at the given

[4] `https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/frame/Frame.html` (visited on 2022-11-21)

index contains an integer value before it is returned to the caller. In case the underlying slot would contain a value with a different type, an exception would be thrown.

While this is extremely useful for dynamically typed languages, statically types languages, such as WebAssembly, already know the types of the Frame slots ahead of execution and validate their bytecode during parsing to guarantee consistent types on the operand stack. Therefore, the static frame API introduces methods optimized for statically typed languages that do not perform type checks when accessing the Frame in the interpreter.

Having no type checks at all yields optimal interpreter performance. However, since the Frame is a public API that can be used by any guest language, having no type safety guarantees could lead to misunderstandings and erroneous behavior.

The Frame stores primitive values and reference values in two different arrays. Without any type checks, this would make it possible to store a primitive value and a reference value in the same Frame slot. While this works in the interpreter, it would result in unexpected return values in the GraalVM compiler, since the differentiation between primitive and reference values is no longer given there. In addition, a language could write an `int` value into a Frame slot and read a `long` value from the same slot. Here, the compiler could not give any guarantees about the upper 32 bits of the long value, which could again result in unexpected return values.

To counteract this behavior, the static frame API introduces type checks based on Java *assertions*. Here, instead of performing the type checks in if conditions and throwing an exception, the type checks are performed behind `asserts` leading to assertion errors when failing. Since assertions can be disabled in production code, the interpreter performance is not affected in those cases, while during development the assertions guarantee type safety. The type information still has to be updated in compiled code, since assertions are not considered during PE. Otherwise, when code is deoptimized, this could lead to inconsistent type information.

The results of the static frame API are an up to 32% improvement of the interpreter performance due to the reduction of array reads and writes. The concrete numbers for our benchmarks can be seen in Section 4.6.

4.4 Memory Access

According to the WebAssembly specification [6], memory addresses are represented by 33-bit unsigned integer values composed of a 32-bit base address and a 32-bit offset. The 33-bit addresses are constructed by adding the base address and offset together. This requires the addresses to be stored as `long` values in GraalWasm. In addition, GraalWasm uses the `ByteArraySupport`[5] API provided by the Truffle framework to interact with the byte array in the `ByteArrayWasmMemory` implementation. Previously, the `ByteArraySupport` was limited to only supporting `int` values as byte array offsets, since byte arrays in Java can only be indexed by signed integer values.

This resulted in GraalWasm having to perform two bounds checks on every memory access. First, it had to be checked that the 33-bit unsigned integer address fits into a 32-bit signed integer. Second, it had to be checked that the resulting 32-bit signed address lies inside the bounds of the memory.

This represents a problem for memory access performed in loops. While duplicate bounds checks have little impact on the interpreter performance, the compiler cannot identify whether bounds checks only depend on the loop-invariant parts of a loop. This is due to the type conversion performed between the two bounds checks. Therefore, the compiler is not able to perform loop-invariant code motion on these bounds checks and the checks are performed on every loop iteration.

To prevent this behavior and to enable loop-invariant motion, we introduced new methods to the `ByteArraySupport` that allow for byte array offsets to be represented by `long` values. This allows having a single bounds check for every memory access.

As a result, these bounds checks are now correctly moved out of loops. This results in peak performance improvements of up to 26% in memory-intensive benchmarks. Detailed numbers can be seen in Section 4.6.

[5] `https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/memory/ByteArraySupport.html` (visited on 2023-01-04)

4.5 Removal of Redundant Data in the Bytecode

To further optimize the memory overhead of GraalWasm, we remove all redundant data after parsing. This includes the data section of the bytecode and the source information provided by the execution environment.

The data section of a module in the WebAssembly bytecode represents the initial data of a memory. This information is read by the parser and preserved by the linker. The linker uses *link actions*, represented by functions, that are created during parsing and executed during linking, since they may rely on data that is not present during parsing, such as imports from other modules. These functions capture the data relevant to their linking step in their closure, which is why the initial data of a memory area is captured in the corresponding link action. This allows for the data section to be dropped from the WebAssembly bytecode after the parsing of a module is completed. In cases where the data section needs to be parsed a second time, such as unit tests, a flag can be passed to GraalWasm to preserve the data section.

We considered removing other parts of the WebAssembly bytecode, since the code section, i.e., the instruction sequences of all the functions, is the only relevant part of the WebAssembly bytecode during run time. The problem is that the code section is at the end of the bytecode right before the data section and all of the other information is located at the beginning of the bytecode. Removing this information would shift the code section in the bytecode. While the instruction sequences of functions are relocatable, debugging information relies heavily on absolute bytecode offsets, which is why we decided to keep the rest of the bytecode as it is. Furthermore, all other sections are very concise, which makes the memory overhead gain negligible.

When an application is executed on GraalWasm the execution environment provides the code of the application in the form of a Truffle *source*. This source holds the contents of the source file of the application, in addition to some meta-data. In GraalWasm the WebAssembly bytecode is extracted as a byte array from this source. After this initial step, the source is no longer needed and can be dropped. In cases where a source is needed at run time, such as debugging, it can be reconstructed from the byte array.

This removal of redundant data in the bytecode reduces the memory overhead by up to 38%. The resulting memory overhead numbers can be seen in Section 4.6.

4.6 Evaluation

To verify that the performance gains of the introduced changes are as expected, the GraalWasm changes were evaluated against a set of micro-benchmarks for interpreter performance, peak performance, and memory overhead. The conceptually expected results and some performance numbers were already stated in the corresponding sections and will be discussed in further detail in the following sections after introducing the used benchmarks and the used setup.

4.6.1 Benchmarks

The selected interpreter benchmarks are a set of popular benchmarks including *deltablue* [19], *fibonacci*, *richards*[6], and an implementation of the *sieve of Eratosthenes* similar to the one introduced in Chapter 3. These benchmarks might not fully reflect the behavior of real-world applications, but focus on certain scenarios encountered in a wide range of applications including a huge number of memory accesses (sieve) or a large number of recursions (fibonacci). All of the interpreter benchmarks are implemented in C and have a fixed problem size.

To measure peak performance, a set of micro benchmarks including *cdf*, *event-sim*, *fft*, and *phong* [20], among others, were selected. They again are written in C, have a fixed problem size, and focus on specific scenarios such as heavy use of recursion (fft) or the simulation of a state-based system (event-sim).

To evaluate the memory overhead, we used the *go-hello* and *doom* benchmarks. The go-hello benchmark represents a simple Go application printing "hello world" to the console. This alone would not represent a fitting memory benchmark. However, at the time of compilation, the Go compiler included a large section of the standard library, which lead to a large number of functions in the benchmark. This allows the evaluation of the internal representation of the function graphs based on this benchmark. The doom benchmark is a WebAssembly port by Diekmann[7] of the video game DooM from 1993. While this benchmark has a smaller set of functions than the go-hello benchmark, it includes a large

[6] `https://www.cl.cam.ac.uk/~mr10/Bench.html` (visited on 2022-12-15)
[7] `https://github.com/diekmann/wasm-fizzbuzz` (visited on 2022-12-17)

set of game assets that represent the initial data of the memory. This allows the evaluation of changes to the internal representation of the initial data in GraalWasm.

The concrete implementations of all used benchmarks, except for the doom benchmark, are available on the GraalVM Github repository[8]. The source files of the interpreter benchmarks can be found in the *vm* directory while the peak performance and memory overhead benchmarks can be found in the *wasm* directory.

4.6.2 Setup

To gather the benchmark results for this thesis, we used a laptop with an Intel Core i5-8365U quad-core processor at 1.6GHz and 16GB of RAM running Zorin OS 16.1. The benchmarks were compiled with the emscripten compiler[9] in version 1.39.13. The interpreter benchmarks were compiled with optimization level O0 due to some limitations in the previous state of GraalWasm, while the peak performance benchmarks were compiled with optimization level O3. The go-hello benchmark was compiled with the Go compiler while the doom benchmark was compiled according to the instructions provided by Diekmann in his GitHub repository.

The benchmarks were executed on the community edition of the GraalVM and JDK 11.0.15. The interpreter benchmarks were executed by disabling the Graal compiler with the help of a feature flag. This allowed us to execute the benchmarks in an interpreter-only mode on the JVM. In addition, the interpreter benchmarks were executed in native mode. Here, the interpreter is ahead-of-time compiled by the GraalVM and is executed as a native binary called *native image*[10]. The native binary is compiled with a different set of optimizations compared to the JVM version. This explains why the performance of both versions differs in our interpreter benchmarks.

The previous state, which represents the implementation of GraalWasm before this thesis, is based on commit `3bb91fc`, the flat bytecode interpreter model is represented by commit `e707bb7`, the static frame API is represented by commit `1c0cf82`, the memory access changes are represented by commit `b64d587`, the removal of redundant data is represented by commit `123490e`, and the introduction of the compact format for the flat bytecode

[8] `https://github.com/oracle/graal` (visited on 2022-12-15)
[9] `https://emscripten.org` (visited on 2022-12-05)
[10] `https://www.graalvm.org/22.0/reference-manual/native-image` (visited on 2022-12-08)

interpreter model is represented by commit `b7a0a29`. The compact data format was introduced after the removal of redundant data. This is why it is listed as a separate entry in the benchmarks.

All interpreter performance benchmarks were executed for 50 iterations in the same process. The first 20 iterations were used for warm-up. This is necessary to get stable performance numbers. The last 30 iterations were used for measurements. The results are extracted by taking the mean and standard deviation of these measurement iterations. The same is true for the peak performance benchmarks, but instead, 6 warm-up iterations and 8 measurement iterations were performed. For the memory benchmarks, 10 warm-up iterations and 6 measurement iterations were performed. The used iteration counts were derived from manual testing to ensure stable results.

4.6.3 Interpreter Performance Results

For the interpreter performance benchmarks, a clear trend was identified based on the results depicted in Figure 4.6. This trend shows that for all benchmarks both the flat bytecode interpreter model (flat.), as well as the static frame API (stat.), had a positive impact on the interpreter performance while the other changes do in general not have a huge impact on the performance at all. However, both the interpreter-only version of the JVM and the native image version of the benchmarks show a slight performance reduction due to the changes in memory access (mem.) and the removal of redundant data (data).

The flat bytecode interpreter model affected the interpreter performance according to our expectations. This can mainly be attributed to the reduction of Java function calls in the interpreter by reducing the number of nodes in a function's AST and the general simplification of the control flow. Resulting from that, an average performance of 15% was gained by this change compared to the previous version of GraalWasm.

Another performance gain was achieved by the static frame API due to the reduction of array accesses in the interpreter. This again corresponds with our predictions and results in an average performance gain of 25% compared to the flat bytecode interpreter model.

While both the changes in memory access and the removal of redundant data lead to a slight decrease in performance, we were not able to link the regressions to the changes introduced in GraalWasm. The changes in memory access simplified the interpreter by

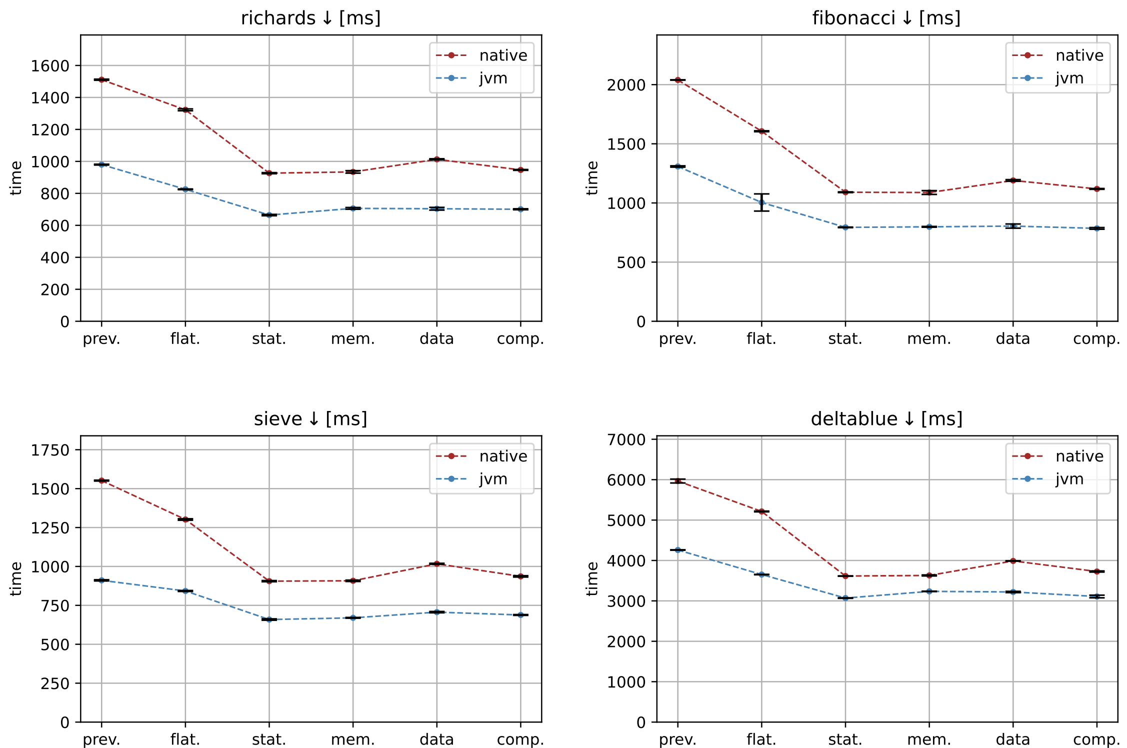


Figure 4.6: Results of the interpreter performance benchmarks of all changes introduced by this thesis. Lower is better, as depicted by the arrows next to the benchmark names.

removing type casts from `long` to `int` and even reduced the code executed for bounds checks on every single memory access. Especially the removal of redundant data does not change the implementation of the interpreter but is limited to a reduction of run-time data. Therefore, possible explanations of the regressions could be changes in the GraalVM or the Truffle framework in the time between implementing the static frame API and the removal of redundant data.

As a result of all the changes implemented as part of this thesis, a 35% increase in interpreter performance in the range of [24.43%, 45.15%] was achieved in our interpreter benchmarks.

4.6.4 Peak Performance Results

In the results of the peak performance benchmarks depicted in Figure 4.7 five types of behaviors can be identified:

- Benchmarks with no significant performance impact such as *phong*.
- Benchmarks only affected by the introduction of the flat bytecode interpreter model such as *cdf*.
- Benchmarks only affected by the changes to memory access such as *digitron* and *strings*.
- Benchmarks affected by both changes such as *event-sim*, *hash-join*, *merge-join*, and *qsort*.
- Benchmarks showing regressions such as *fft*.

The phong benchmark represents an implementation of the phong shading algorithm. It is mainly bound by a lot of matrix operations and does not require many memory accesses. This is why no significant improvements can be achieved by either the flat bytecode interpreter model (flat.) or the changes to memory access (mem.).

The cdf benchmark is performing calculations in a small number of functions that do not require any memory access. The introduction of the flat bytecode interpreter model reduced the complexity of these functions from the perspective of the compiler which allows for more optimizations to be applied by the compiler in comparison to the previous version. This explains the performance gain by the flat bytecode interpreter model, while the changes in memory access do not have any effect on this benchmark.

Both the digitron and the strings benchmark perform a lot of memory interactions including building up expression trees for mathematical expressions in memory or copying a lot of strings. This is, why they mainly benefit from the changes to memory access. The digitron benchmark is very unstable in general, which is why we see a drop in performance with the introduction of the flat bytecode interpreter model and an improvement with the introduction of the compact extra data format. Especially the latter case does not have any impact on the peak performance of this benchmark, since the values of the extra data array are seen as constants in compiled code and are therefore not bound by

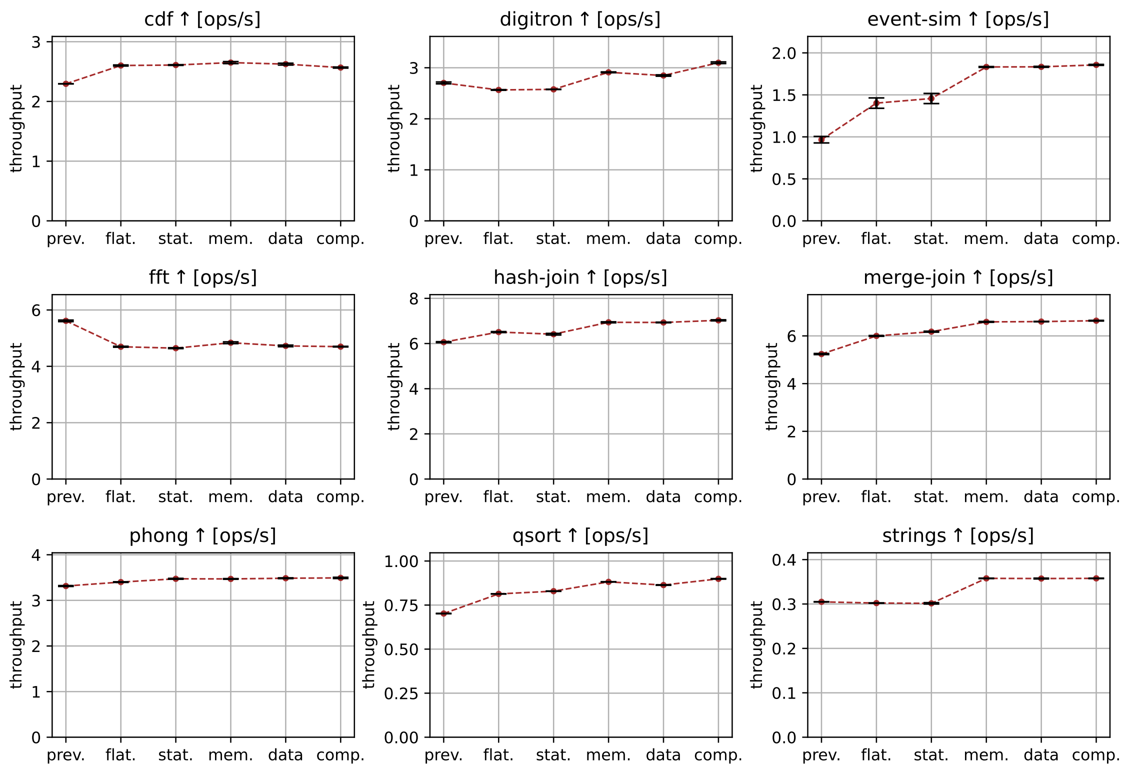


Figure 4.7: Results of the peak performance benchmarks of all changes introduced by this thesis. Higher is better, as depicted by the arrows next to the benchmark names.

the required number of array reads on the extra data array. This is why we conclude that the improvement in the digitron benchmark is not caused by this change to GraalWasm but can be attributed to the unstable nature of the benchmark.

The event-sim, hash-join, merge-join, and qsort benchmarks implement a state-based event system and different array sorting algorithms. They all profit from simplified control flow and faster memory access, which is why we see improvements in both cases.

To identify the cause of the performance regression in the fft benchmark we investigated the GraalVM runtime graph structures of the benchmark with the help of the *Ideal Graph Visualizer*[11]. While both, the graph of the previous version and the graph of the flat bytecode interpreter model, were identical, we were able to identify changes in the inlining

[11] `https://ssw.jku.at/General/Staff/TW/igv.html` (visited on 2022-12-08)

decisions performed by the compiler. More functions were inlined compared to the previous version of GraalWasm, which caused negative effects on the peak performance of the fft benchmark.

According to Scheifler [21], inlining is an optimization technique where a function call is replaced by a modified version of the body of the called function. The GraalVM performs inlining based on some heuristics and predefined parameters. One of these parameters is the *maximum inlining depth*. A maximum inlining depth of two for example would mean, that function calls inside inlined functions can again be inlined into the main function. Defining a maximum inlining depth of one would not allow this behavior, but only limit the compiler to inlining function calls that are directly in the main function. Inlining is one of the most profitable optimization techniques but can be harmful to recursive algorithms according to our observations.

Manually setting the maximum inlining depth of the GraalVM compiler to a lower limit such as two or one improved the performance of the fft benchmark up to a point, where it even outperformed the results of the previous implementation. The same is true for other recursion-based benchmarks such as the sorting algorithms in the merge-join and hash-join benchmarks.

While the impact of the changes introduced by this thesis depends heavily on the tasks performed by concrete benchmarks, a general improvement of the peak performance can be identified. The most noticeable changes are the 16% regression on the fft benchmark and the 92% improvement on the event-sim benchmark. The average performance gain is 22% within a range of [-16.30%, 92.23%].

4.6.5 Memory Overhead Results

The results of the memory overhead benchmarks depicted in Figure 4.8 clearly show the impact of the flat bytecode interpreter model (flat.), the removal of redundant data (data), and the introduction of the compact extra data format (comp.). While the most influential change differs for the two benchmarks, a clear improvement in both cases can be identified.

The go-hello benchmark mainly benefits from the introduction of the flat bytecode interpreter model due to its high number of functions. In addition, these functions often

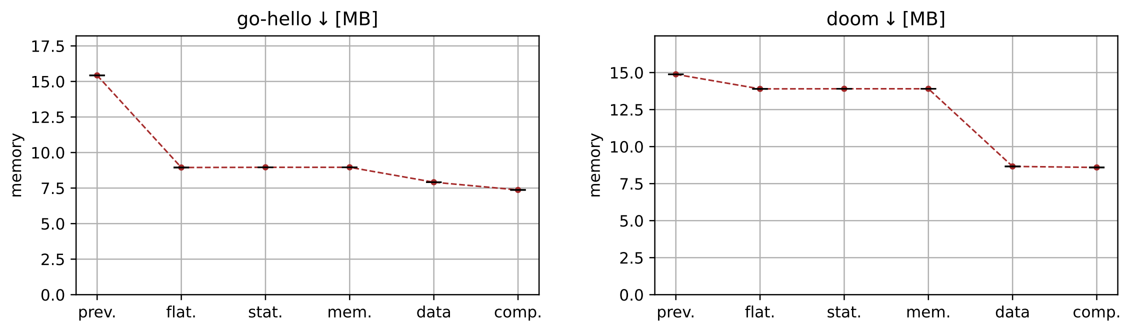


Figure 4.8: Results of the memory overhead benchmarks of all changes introduced by this thesis. Lower is better, as depicted by the arrows next to the benchmark names.

contain a lot of complex control flow. This also explains the impact of the compact extra data format, since the data needed for the control flow is again reduced. While an impact of the removal of redundant data can be seen, the impact is smaller compared to the doom benchmark, since the go-hello benchmark does not define a substantial amount of initial data.

In contrast to that, the doom benchmark, defining a lot of game assets as initial data, profits mainly from the removal of redundant data. The introduction of the flat bytecode interpreter model and the compact extra data format show a smaller impact due to the limited number of functions in the doom benchmark.

As a result of all three changes, an average memory overhead reduction of 47% was achieved in the range of [42.26%, 52.23%].

5 Tooling

Tooling defines a broad category of helper programs that support software developers in writing their applications. While this includes everything from compilers and linkers through formatters and linters up to CPU- and memory profilers, this thesis focuses on further improving the debugging experience in GraalWasm.

It is important to state that the debugging support itself was implemented as a project previous to this thesis and will therefore not be explained in detail. The contributions made by this thesis are the introduction of a small testing framework built on the existing instrumentation support provided by the GraalVM as well as adapting the debugging support in GraalWasm to the changes introduced by the flat bytecode interpreter model as previously explained.

Debugging WebAssembly can be categorized into *bytecode-level debugging* and *source-level debugging*. While bytecode-level debugging allows stepping instruction by instruction through the WebAssembly text format representation of an application, source-level debugging allows stepping through the statements and expressions of the application in the source language that was compiled to WebAssembly. GraalWasm focuses on the latter one.

Most of the languages compiled to WebAssembly, such as C or C++, already support debugging on native target architectures such as x86 or arm64. This already allows developers to detect defects in their applications. Nonetheless, when writing code for other target architectures, such as WebAssembly, it is useful to be able to debug an application in this environment due to nuances in the translation of the source language to the target architecture. An example would be that overflows can happen on one architecture but not on the other based on the sizes of primitive data types.

Since C and C++ represent the main source languages for WebAssembly applications running on GraalWasm, the GraalWasm debugger focuses on the DWARF Debugging

Format [9]. This format encodes all type information about the data types and function types of an application, the locations of variables on the local stack and the heap, and a mapping from source code line numbers to bytecode locations in the WebAssembly bytecode. It is attached to the end of a WebAssembly module in what is defined as custom sections. These sections have a name and a sequence of bytes that are not interpreted by the WebAssembly bytecode parser. This allows them to be ignored when no debugger is attached and enables the lazy loading of debugging information at run time.

A depiction of the debugger running inside the DevTools of the Microsoft Edge browser can be seen in Figure 5.1. Here, a small application is paused at the breakpoint at line 11. The right-hand side of the Figure shows all currently active breakpoints and all the variables in both the global and the local scopes of the application. In addition, the current call stack, containing the *main* and *mul* methods, is depicted.

Figure 5.1: DevTools of the Microsoft Edge browser showing the debugging state of a small application.

5.1 Debugger Adaptation

The Truffle instrumentation API is based around AST nodes. Resulting from that, to set a breakpoint in the source code of an application, a node in the AST representing the statement at the given line has to exist. Since GraalWasm is implemented as a bytecode interpreter loop, nodes for statements do not exist. Therefore, to adhere to the instrumentation API, artificial *statement nodes*, mapping bytecode locations to source code line numbers, have to be generated based on the information in the debugging sections. When the bytecode interpreter loop is executed with a debugger attached, it is checked if the current bytecode offset corresponds to the bytecode location in one of the statement nodes. If this is the case, the node is presented to the instrumentation API such that the debugger stops at the given breakpoint based on the line number in the statement node.

In the version of the debugger previous to the flat bytecode interpreter model, every block node held statement nodes for all possible source code lines of the WebAssembly function they belonged to. Creating only the necessary statement nodes for every block node would have resulted in defining potentially a lot of bytecode ranges. These bytecode ranges would have represented those instructions of a block node, which are not part of any of its child nodes. Resulting from that, all possible source code lines would have to be iterated for every block node and the ones corresponding to the bytecode locations of one of the ranges would have been picked. This would have represented a huge run-time overhead since the debugging information is lazily generated. As a result, a lot of the statement nodes existed multiple times.

In addition, a *debugging state* was needed for every function. This debugging state represented the current line number and current position in the list of statement nodes. The purpose of the debugging state was to prevent hitting breakpoints multiple times when entering a new block node since multiple bytecode locations can map to the same source code line.

Through the introduction of the flat bytecode interpreter model, both the duplication of statement nodes and debugging states are no longer necessary. Since every function is represented by a single node in the flat bytecode interpreter model, the statement nodes are generated once for the entire function and no debugging state is necessary to synchronize the state throughout multiple block nodes.

5.2 Debugger Testing

To test the correctness of the debugger implementation, a small testing framework that abstracts away the implementation details of the Truffle Debugger API[1] was implemented. It allows checking that the debugger stops the running application if a breakpoint is hit, that local and global variables are correctly extracted from their memory locations and that the debugger correctly reports function entries and exists.

To achieve these goals, the introduced debugging framework allows the definition of a mapping between source code line numbers and actions, in the form of functions, that should be performed when the debugger hits the statement at the given line number. The actions receive a reference to the current local variables and, if the source language defines a global scope, access to the variables of the global scope. This allows checking if the local and global variables are correctly mapped to memory locations, based on the debugging information provided by the source language. Furthermore, it allows checking if values are correctly extracted based on their value type encoding, such as float numbers.

In addition, the framework allows the definition of a sequence of source code lines that has to be reached by the debugger in the given order. This allows checking if the line maps in the debugging information, mapping bytecode positions in the WebAssembly bytecode to line numbers in the source code, are correctly extracted.

With this framework in place, we defined a set of scenarios that should be tested to check if the debugger behaves correctly. The scenarios include *arrays*, all kinds of *primitive values*, *bit fields*, *structs* and *classes*, control flow such as *loops* and *recursion*, and many others. The framework exposed some existing bugs in the debugger implementation that were fixed as part of this thesis.

[1] `https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/debug/Debugger.html` (visited on 2022-11-29)

6 Language Features

Compared to other bytecode formats, such as the Java Bytecode [12] or the Common Intermediate Language [13], WebAssembly is a fairly new bytecode format. Therefore, new language features are added regularly. These features evolve through the WebAssembly standardization process and eventually become part of the standard. Throughout the implementation of this thesis, the WebAssembly specification was updated from version 1.0 to version 2.0 including new features such as support for multiple return values or the introduction of reference types. For a WebAssembly runtime, such as GraalWasm, to stay compliant with the specification, these new features have to be implemented.

Some of the features that are part of version 2.0 were already part of GraalWasm while the *fixed-width SIMD* proposal was defined to be out of scope for this thesis. As a result, we decided to implement the *multi-value* proposal, described in Section 6.1, the *bulk memory operations* proposal, described in Section 6.2, and the *reference types* proposal, described in Section 6.3, as part of this thesis. In addition, the *memory64* proposal, described in Section 6.4, was implemented due to a request by another team working on the GraalVM.

The correctness of the implemented features was validated with the help of the *official test suite*[1] provided by the WebAssembly standard. Since at the time of implementation, the official test suite did not yet include tests for the memory64 proposal, a set of basic unit tests were implemented to verify the correctness of this feature.

[1] `https://github.com/WebAssembly/spec/tree/main/test` (visited on 2023-01-24)

6.1 Multi-value Proposal

According to the multi-value proposal overview document[2], allowing for only a single return value per function imposes an artificial arity restriction in stack-based languages. Therefore, a W3C Community Group evolving the WebAssembly specification decided to lift this restriction with the introduction of the multi-value proposal[3].

6.1.1 Specification

The WebAssembly specification [6] after the inclusion of this proposal allows functions to have an arbitrary number of return types, for each scope to define multiple parameters and return values, and for instructions to return an arbitrary amount of values. While lifting the restriction on the number of function return values is motivated by enabling the decomposition of struct types and tuples, having parameters and return types for scopes allows for loop variables to live exclusively on the operand stack without needing an additional local variable, for example. In addition, instructions such as an add operation with an additional overflow indicator value can now be implemented.

To illustrate the benefits of this proposal, the running example introduced in Chapter 3 is updated. First, the output of the function is extended to not only return if the given number is prime or not but to also return how many prime numbers up to the given number exist. An input of 7, for example, would return `true` and 4, since 7 is a prime number and there are 4 prime numbers from 2 to 7 (2, 3, 4, 7). The adaptation of the pseudo code to this change can be seen in Listing 6.1.

Previous to the introduction of the multi-value proposal, this function would have required storing the results in a location in memory and returning the memory address. The caller would then have to extract the values from memory to work with it. If we think about an implementation of the pseudo code in C we would need to return the values in a struct. In WebAssembly, this struct can now be decomposed into multiple return values and therefore does not require any memory allocation.

[2]`https://github.com/WebAssembly/multi-value/blob/master/proposals/multi-value/Overview.md` (visited on 2022-12-11)
[3]`https://webassembly.github.io/multi-value/core/_download/WebAssembly.pdf` (visited on 2023-01-24)

Listing 6.1: Pseudo code of the sieve of Eratosthenes algorithm including multi-value extensions.

```
1   function sieve:
2       input:  integer n where n > 1.
3       output: whether n is prime or not and the number of prime numbers
            up to n.
4
5       let M be an array of boolean values from 2 to n, all set to true.
6       let x be n - 1.
7
8       for i in 2, 3, 4, ... to sqrt(n):
9           if M[i] is true:
10              for j in 2 * i,  3 * i, 4 * i, ... to n:
11                  if M[j] is true:
12                      M[j] = false
13                      x--
14      return M[n], x
```

6.1.2 WebAssembly Implementation

The implementation of parameters and return values for scopes did not require any changes in the interpreter, but only in the validation step of the bytecode parser. Since `block` scopes did already support a single return value in GraalWasm, this logic was extended to multiple return values and was adopted for `if` and `loop` scopes. Parameters are passed by the enclosing scope to the inner scope via the operand stack, which again does not require any changes in the interpreter.

Java does not support multiple return values, representing a challenge for the implementation of the multi-value proposal. In addition, GraalWasm is one of the few language implementations on the GraalVM that, apart from growing memory and tables, does not allocate any Java objects after parsing is finished. This significantly reduces the garbage collection pressure, improving performance. To preserve this property, our first approach of packing the result values into a Java object array that gets returned by a multi-result function represents a sub-optimal solution. The approach would introduce a lot of short-lived objects that produce garbage every time a function with multiple return values is called.

Therefore, the multi-value proposal is implemented with the help of an array called the *shadow stack*. This array exists once for every execution thread and can grow dynamically based on the needed number of return values. While this can still lead to situations where every function call grows the shadow stack and therefore produces a lot of garbage, this only happens at most once per function and the length of the shadow stack should in general stabilize after central functions of an application have been called. Furthermore, the shadow stack only grows in interpreted code, but never in optimized machine code.

To indicate that the return value of a function can be found on the shadow stack, the root node of a function returns a special *multi-value result*. Based on this, the caller of a function can decompose the values from the shadow stack onto its operand stack.

6.1.3 JavaScript Implementation

Since the shadow stack only exists in the context of GraalWasm, when a multi-value result is returned to an external caller, such as a JavaScript function, the shadow stack has to be materialized. This materialized version of the shadow stack is represented by an array-like object implementing the `TruffleObject` interface provided by the GraalVM Polyglot API. The array-like object is decomposed by Graal.js and every value is transformed from a WebAssembly representation to a JavaScript representation.

If a multi-value result is returned by a call from WebAssembly to a JavaScript function, Graal.js again transforms the values from a JavaScript representation to a WebAssembly representation and packs them into an array-like object that can be decomposed by GraalWasm. In these cases, the values are not put onto the shadow stack, but directly onto the operand stack of the calling WebAssembly function.

6.2 Bulk Memory Operations Proposal

Memory operations such as copying large chunks of memory or initializing parts of the memory with a predefined value can be found in a multitude of applications. This is why a W3C Community Group decided to create specific WebAssembly instructions for these

operations in the bulk memory operations proposal[4]. This allows runtimes to provide efficient operations, replacing implementations in WebAssembly bytecode.

6.2.1 Specification

The bulk memory operations proposal introduces multiple instructions for efficient memory management to the WebAssembly specification [6]. It extends the initialization of memory and adds additional methods for dealing with large chunks of memory.

Previous to the bulk memory operations proposal, the initial data of memory areas was represented by data sections and was copied when a module was instantiated. The bulk memory proposal introduces a way of dynamically initializing memory at run time. To support this behavior, data sections are split up into active sections, initialized when a module gets instantiated, and passive data sections, potentially initialized at run time. This allows loading certain chunks of data only when they are needed by the application. Since passive data sections could potentially exist, even after they are used for initializing parts of the memory, the bulk memory proposal also introduces a way of explicitly discarding them via the *drop* instruction, when they are no longer needed, to free up memory.

Furthermore, the bulk memory operations proposal introduces a *copy* and a *fill* method for memory areas. They allow copying data efficiently from one part of the memory to another or initializing parts of the memory with a given value.

In the context of our running example, the initialization of the collection holding the boolean values can now be implemented with a single fill instruction. Previously, this would have required a separate function that individually sets every memory location to `true` in a loop.

6.2.2 WebAssembly Implementation

To support passive data sections, references to the data extracted from the bytecode were added to the instance representation in GraalWasm. When the corresponding passive sections are dropped, they are set to `null`. Unfortunately, this does not free up

[4] `https://webassembly.github.io/bulk-memory-operations/core/_download/WebAssembly.pdf` (visited on 2023-01-24)

memory, since the passive sections are kept by the linker, in case the module is instantiated multiple times. The added instructions were implemented as methods on both memory implementations.

One of the main goals of implementing new features in GraalWasm is that they must not negatively impact the performance of existing WebAssembly applications. After adding the new bulk memory instructions to the interpreter and evaluating the interpreter performance with the benchmarks introduced in Section 4.6, we saw a clear regression in the native version of the interpreter benchmarks.

Investigating the problem showed that loop unrolling was no longer performed when reading constants from the bytecode, due to reaching some complexity threshold in the interpreter. These constants are represented by LEB-128 values with a maximum size of 32 bits. Since the LEB-128 encoding uses the first bit of every byte as a continuation indicator bit, reading the constants in the bytecode was implemented with a loop that exits when the indicator bit is zero.

Since the constants are restricted to 32-bit values, we decided to manually unroll the corresponding loops by copying their loop bodies five times, since this is the maximum amount of iterations possible for a 32-bit LEB-128 value. While manual loop unrolling is in general not advisable, it allowed us to get back to the previous interpreter performance.

6.3 Reference Types Proposal

While the numeric types defined by the WebAssembly type system suffice to represent all different types of applications, it limits the interoperability with embedding environments. For example, if a JavaScript function wants to pass an object to a WebAssembly function, it can only do so by having a lookup table on the JavaScript side that maps objects to numeric values. Due to this limitation, a W3C Community Group introduced reference types for better interoperability with host environments in the reference types proposal[5].

[5] `https://webassembly.github.io/reference-types/core/_download/WebAssembly.pdf` (visited on 2023-01-24)

6.3.1 Specification

Version 2.0 of the WebAssembly specification [6] defines reference types as *opaque* pointers. This implies that WebAssembly applications cannot directly interact with the objects behind the references but can only pass them to functions implemented by the embedding environment. In addition, references cannot be stored in WebAssembly memory, but only in WebAssembly tables. In the reference types proposal, the tables were extended from only holding references to functions, to now being able to hold references to arbitrary objects.

Furthermore, the reference types proposal introduces the *externref* type, indicating a reference to an object provided by the embedding environment. The externref type, in addition to the already existing *funcref* type, can now be used for value types, such as the return values of functions or the parameters of instructions. This gave rise to the introduction of additional table methods such as getting and setting individual entries, retrieving and extending the size of a table, and the introduction of a *fill* and a *copy* method for tables, similar to those explained for memory in Section 6.2. In addition, the initialization process and the introduction of dynamically loading parts of the data at run time, as explained in Section 6.2, were adopted for tables.

Furthermore, the proposal lifts the restriction on the number of tables per module. Previously every module was limited to a single table. This led to scenarios where certain functions were exposed to the embedding environment through the export of a table, although they were only needed internally. The reference types proposal now allows having a set of functions that are public and can be exported to the embedding environment, while other functions might be private and should only be accessible from within a module, by having two separate tables.

To give a concrete example of the usefulness of this proposal, the running example from Chapter 3 is adopted with support for collections. This requires three changes. First, a new function *sieve_multi* is introduced that takes a reference to a collection c and the size of the collection and calculates for all entries in c if the value is prime or not. It writes the output of the *sieve* function to the same entry in c. Second, a function *get* for reading an entry from the collection has to be provided as an import by the embedding environment. Third, a function *set* for writing the output to an individual entry has to be provided in the same way by the embedding environment. We cannot provide the get and set

methods as parameters to sieve_multi since WebAssembly currently does not support calling functions provided as parameters.

The adaptation of the pseudo code can be seen in Listing 6.2. The same could be achieved by a method in the embedding environment, but for the sake of this example, we implement the functionality in WebAssembly.

Listing 6.2: Pseudo code of the sieve of Eratosthenes algorithm including reference types extensions.

```
1   function sieve:
2       input:  integer n where n > 1.
3       output: whether n is prime or not and the number of prime numbers
            up to n.
4
5       let M be an array of boolean values from 2 to n, all set to true.
6       let x be n - 1.
7
8       for i in 2, 3, 4, ... to sqrt(n):
9           if M[i] is true:
10              for j in 2 * i, 3 * i, 4 * i, ... to n:
11                  if M[j] is true:
12                      M[j] = false
13                      x--
14      return M[n], x
15
16  function sieve_multi:
17      input:  collection c holding values that should be checked and size
             s where s is the number of entries in c
18      global: function get for reading an entry from c and function set
            for writing an entry in c
19      output: updated entries in c
20
21      for i in 0, 1, 2, ... to s:
22          n = get(c, i)
23          p, x = sieve(n)
24          set(c, i, p, x)
```

Since we use JavaScript as our embedding environment, we can provide an object array to the sieve_multi function. The objects can hold a *value*, an *isPrime*, and a *numberOfPrimes* property. The get and set methods read the value property and write the isPrime and numberOfPrimes properties of the objects in the array. The result of calling sieve_multi is

a collection of objects holding integer values, information about whether they are prime or not, and the number of prime numbers up to the integer value.

6.3.2 WebAssembly Implementation

To add support for passive element sections, equivalent to the passive data sections on memory explained in Section 6.2, the data of the elem segments, containing the indices of functions that should be stored in a table, are transformed into a Java object array that holds references to the functions themselves and is stored in a WebAssembly instance. Furthermore, the newly introduced table methods were added to the GraalWasm table implementation, and support for multiple tables was added to the instances.

The challenging part of the implementation of the reference types proposal was the adaptation of existing operations to support reference types while keeping the interpreter performance stable. As explained in Section 4.3, the Frame stores primitive and reference values in different ways in the interpreter. To support this behavior, the static frame API introduced different methods for dealing with primitive values, reference values, and scenarios where it is not known if the value is a primitive or a reference value. An example would be the *copy* method. It exists as `copyPrimitiveStatic`, `copyObjectStatic`, and `copyStatic`.

In GraalWasm, both locals and operand stack values are stored in the Frame. The WebAssembly instructions `local.get`, for getting the value of a local variable, `local.set`, for setting the value of a local variable, and `local.tee`, for setting the value of a local variable while keeping the value on the operand stack, are implemented by copying a value from one Frame slot to another. Previous to the reference types proposal this was implemented with the `copyPrimitiveStatic` method, resulting in one array read and one array write for each access to a local variable.

With the introduction of reference types, the first approach was to move to the `copyStatic` method for all accesses to local variables. This resulted in a performance regression in both versions of our interpreter benchmarks, since the Frame performs copy operations on both its primitive array and reference array resulting in two array reads and two array writes for each copy operation.

Since local variables have a predefined value type in WebAssembly, our second approach was to use this information to call the appropriate copy method at run time. Since the type information for locals was already stored in an array in the function node, to correctly initialize local slots in the Frame, we tried a lookup of this information for every access to a local variable. While this improved the interpreter performance, it was still worse than the previous version and, therefore, did not represent an optimal solution. The problem of having additional array accesses remained and was moved from both arrays being in the Frame to one array in the Frame and one array in the function node. Instead of having four array accesses to two arrays, we now had three array accesses to two arrays.

As a result, we identified that the core of the problem was that the type information of local variables was checked at run time, although it is statically known, resulting in additional array reads. This is why the final approach introduces custom bytecode instructions for primitive and reference values. The WebAssembly specification [6] defines a set of opcodes reserved for future use. We used five of them to implement reference-type versions of the `get`, `set`, and `tee` instructions on local variables and the `select` and `drop` instructions. The latter two also have to differentiate between primitive types and reference types due to the introduction of the reference types proposal. The instructions replace the original opcodes in the GraalWasm parser by checking the actual value types of the original instructions. This resulted in a similar interpreter performance to the previous GraalWasm version regarding local variable access.

The approach of using reserved opcodes is not optimal, since it is very likely that they are used by new language features in the near future. A solution for this problem could be the introduction of a custom run-time bytecode format, as described in Chapter 8.

The same differentiation between primitive and reference values had to be made for copying result values in the unwind process explained in Section 4.2. This meant that every extra data entry associated with an instruction that performs a stack unwind needed an indicator, stating the types of its return values. Storing the value type of every individual return value was not possible, since the size of every extra data entry had to be fixed for fast lookup. Therefore, we decided to have one return type indicator for all the return values of an extra data entry.

In the compact extra data format, the return type is indicated by two individual bits *t0* and *t1*. Primitive types are indicated by $t0 = 1$ and $t1 = 0$, reference types are indicated

by $t0 = 0$ and $t1 = 1$, and a mixture of both is indicated by $t0 = 1$ and $t1 = 1$. To store this information, the result count and the stack size had to be reduced to 7 bits instead of 8 bits. The leading bits of both values are used to encode the type of the result values as depicted in Figure 6.1. In the extended format, an additional integer value was added for the type information using the same two-bit encoding as in the compact format.

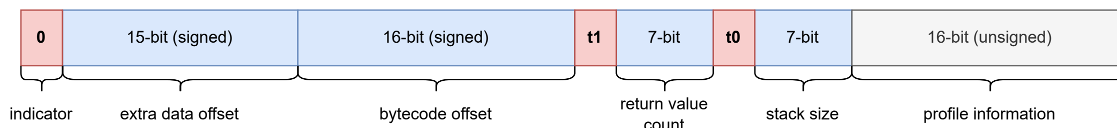


Figure 6.1: Extension of the compact format of a `br_if` instructions with return type indicator bits.

As a result, the unwind operations can be performed based on the type information provided by the extra data array. This allows for an interpreter performance similar to the previous version of GraalWasm.

6.3.3 JavaScript Implementation

Due to the polyglot functionality of the GraalVM, implementing the interoperability of the reference types proposal in Graal.js and GraalWasm is straightforward. Since both runtimes are implemented in Java, objects can be shared without any need for a transformation.

The challenging part of this proposal was the handling of null values. The polyglot API does not support Java null values for any of its methods. Therefore, every runtime has to implement custom null objects. These objects have to implement the `TruffleObject` interface and have to override the `isNull` method.

In the reference types proposals, references can be null, which is represented by the null object provided by GraalWasm. Null values can also be provided by JavaScript. For example, through function calls. While JavaScript has two null values, *null* and *undefined*, according to the specification [3], only one of them, null, is translated to the WebAssembly null object, while the other is seen as an arbitrary reference. However, when passing the Graal.js implementations of null or undefined to GraalWasm through the polyglot

API, they can no longer be differentiated, since both of their `isNull` methods return `true`. Therefore, a way to differentiate them had to be found.

Our first approach was that every function call from Graal.js to GraalWasm, which could potentially have to check for null values, had to provide the Graal.js null object as a parameter. On the GraalWasm side, a reference equivalence check was used to identify null values. This resulted in a huge number of null checks on the GraalWasm side, although the actual value was already known on the Graal.js side. In addition, this solution was not flexible enough, since it only supported a single null value to be translated to the GraalWasm null object. It could happen that languages other than JavaScript would translate all of their null values to the GraalWasm null object.

As a result, we decided to perform the null checks on the Graal.js side. For this to work, GraalWasm had to expose its null object via the polyglot API with the help of a *ref_null* method. With this in place, Graal.js uses its custom null object implementation to check for null values and passes the WebAssembly null object to GraalWasm. The Graal.js undefined object is passed as is, allowing null and undefined to be differentiated.

6.4 Memory64 Proposal

While for most applications 4 gigabytes of memory is sufficient, there is currently no way of addressing more than this given amount in a WebAssembly application. Therefore, a W3C Community Group decided to introduce the memory64 proposal. The memory64 proposal is currently in stage 3 of the standardization process and does not provide any official changes to the specification yet. Therefore, its current definition can only be found in its overview document[6]. It extends the memory addresses in WebAssembly from 32 bits to 64 bits and removes the current addressable memory limitation of 4 gigabytes.

6.4.1 Specification

The memory64 proposal introduces a new *index type* that is used by both memory and tables to identify their address width. The index type can either be 32 bits or 64 bits. While

[6] `https://github.com/WebAssembly/memory64/blob/main/proposals/memory64/Overview.md` (visited on 2022-12-18)

memory can use both values, tables are still limited to 32 bits. In addition, all memory operations now use operands based on the index type of a memory instance, including loading and storing values.

6.4.2 WebAssembly Implementation

While both changes to the validation process and the interpreter were straightforward, the difficult part of this proposal was the implementation of memory in Java that supports more than 4 gigabytes of data.

Both the byte array used by the `ByteArrayWasmMemory` and the `ByteBuffer` used by the `UnsafeWasmMemory` are limited to signed 32-bit index values and, therefore, to a theoretical upper limit of 2 gigabytes of data. The actual limit is a little bit lower due to additional information needed for object headers and length information.

For the unsafe version of the memory, we introduced a `NativeWasmMemory`. This no longer uses a `ByteBuffer`, but instead directly allocates memory via `Unsafe`. This uses a `long` value for the size and, therefore, provides enough space for the memory64 proposal.

Our attempt for a byte-array-based memory with more than 2 gigabytes of memory was using a multi-dimensional byte array. Although this would be sufficient in terms of memory size, measuring the performance of the resulting implementation with the help of the peak performance benchmarks resulted in performance numbers, which were orders of magnitudes worse than the `ByteArrayWasmMemory`. Thus, we decided to limit the support of 64-bit memory addresses to the unsafe memory implementations.

JOHANNES KEPLER
UNIVERSITY LINZ

7 Related Work

7.1 Sulong

Sulong [22] is a system for executing LLVM-based languages such as C and Fortran on the JVM. By leveraging optimizations from both static (LLVM) and dynamic compilation (GraalVM) it reaches a peak performance comparable with results from pure static compilers. Sulong implements an LLVM IR bitcode interpreter based on the Truffle framework allowing for seamless interoperability with other Truffle languages. In contrast to the bytecode interpreter in GraalWasm, Sulong encodes sequential instructions as an AST and uses a dispatch node for control flow to deal with the possibly unstructured control flow in the LLVM IR. Since most of the languages compiled to WebAssembly can also be compiled to LLVM IR, Sulong represents an alternative to GraalWasm on the GraalVM.

7.2 GraalSqueak

Niephaus et al. [14] compare an AST-based and a bytecode-interpreter-based implementation of their Squeak/Smalltalk interpreter, called *GraalSqueak*, in the context of the Truffle framework. Their AST-based approach transforms every bytecode into an AST node and reconstructs loops to reach better performance. In addition, they implemented a bytecode interpreter loop, similar to GraalWasm. However, their approach uses a predefined set of control-flow nodes instead of direct jumps in the bytecode. By comparing both implementations, they found that the bytecode-interpreter-based implementation requires nearly no warmup time compared to the AST-based approach. However, since they only compare two micro-benchmarks, these results might not be generalizable. Their work highly influenced this thesis, by providing a basis for the flat bytecode interpreter model and how to optimize bytecode interpreters in the context of the Truffle framework.

7.3 TruffleWasm

TruffleWasm [23, 24] is a standalone WebAssembly runtime built using Truffle and the GraalVM similar to GraalWasm. However, instead of using a bytecode interpreter loop, all WebAssembly instructions in TruffleWasm are represented by individual Truffle nodes. Control-flow in TruffleWasm is implemented similarly to the previous version of the GraalWasm interpreter by having specific Truffle nodes for loops and conditional statements, but instead of using return statements to branch out of scopes, they use `ControlFlowExceptions` provided by the Truffle framework. As already stated in their work, this can lead to an expensive number of exception catches and rethrows and, therefore, to sub-optimal interpreter performance. Similar to GraalWasm, TruffleWasm uses two different memory implementations, one based on a ByteBuffer, similar to the `ByteArrayWasmMemory`, and one using native memory allocation, similar to the `NativeWasmMemory` introduced in Section 6.4. TruffleWasm supports WASI and version 1.0 of the WebAssembly specification. Unfortunately, no executable or source code of TruffleWasm is available, which is why we were not able to compare the performance of GraalWasm and TruffleWasm.

8 Future Work

The goal of this thesis was to improve all performance aspects of GraalWasm, to provide better tooling support, and to implement some of the new language features of WebAssembly 2.0. While all of these aspects were improved as part of this thesis, there is still room for improvement regarding the general performance and the completeness of tooling and language features.

The implementation of new language features in Chapter 6 showed that custom adjustments to the WebAssembly bytecode are inevitable to keep performance stable in GraalWasm. As a result, a fully-custom run-time bytecode format would be a useful addition to GraalWasm in the future. It would allow handling GraalVM-specific features, such as instructions differentiating between primitive and reference values, more easily. In addition, the currently separate *extra data arrays* could be merged into the custom bytecode format for better cache locality and for an even more concise representation of run-time data. This would potentially further improve interpreter performance and reduce the memory overhead of GraalWasm.

In terms of tooling, debugging support could be extended to further languages. As a first step, adjustments to the implementation of the current debug format would have to be made, to support other LLVM-based languages including Rust. In addition, other debugging formats could be supported to open the platform for a wider range of languages. Furthermore, GraalWasm is currently tightly coupled with Graal.js. However, the GraalVM supports a multitude of languages that can interact with each other. Therefore, the interoperability with other languages could be further explored.

Lastly, GraalWasm is still not fully compliant with the current WebAssembly 2.0 standard. This would require the implementation of the SIMD proposal. In addition, language features of earlier standardization stages could be implemented to have them ready, when new WebAssembly versions are officially released.

JOHANNES KEPLER
UNIVERSITY LINZ

9 Conclusion

This thesis presented the adaptation and extension of an existing WebAssembly runtime based on the GraalVM and the Truffle language implementation framework. It focused on the three software development usability aspects performance, tooling, and language features to improve GraalWasm.

Performance, including interpreter performance, peak performance, and memory overhead, was improved by introducing the flat bytecode interpreter model, the static frame API, the improvement of memory access, and the removal of redundant data in the bytecode. The flat bytecode interpreter model changed the way how control flow is represented and implemented with the help of an extra data array instead of the previous AST-based model. This allows for a faster and more compact representation of applications in GraalWasm. The static frame API allowed us to further improve the interpreter performance by reducing redundant array reads when accessing operand stack values and local variables. Memory access was improved by reducing the number of bounds checks needed for accessing the linear memory used in WebAssembly applications while the removal of redundant data further reduced memory overhead. The evaluation of the added features showed a general improvement in all three performance areas, although also some regressions on recursion-based benchmarks.

In addition, tooling support was improved by adopting the existing debugger to the flat bytecode interpreter model and by implementing a testing framework to verify the correctness of the debugger implementation. The latter allowed testing the correct interpretation of the debugging format and the correct extraction of values from the WebAssembly memory based on their data types. In addition, it allowed checking that the debugger hits breakpoints in the correct order. It even exposed existing bugs that were fixed as part of this thesis.

As a final step, this thesis integrated four new WebAssembly proposals into GraalWasm. The multi-value proposal, the bulk memory operations proposal, the reference types proposal, and the memory64 proposal. The multi-value proposal allows functions to return multiple values and improves scopes and instructions by adding more flexibility in terms of parameters and return values. The bulk memory operations proposal makes dealing with large chunks of memory easier by introducing new ways of initializing and manipulating memory. The reference types proposal introduces new opaque pointers to the language that improve the interoperability with other languages. This allows WebAssembly applications to deal with arbitrary objects provided by their embedding environment and extends the number of use cases for WebAssembly tables. The last proposal, the memory64 proposal, extends the existing 32-bit addresses to 64 bits and allows for memory larger than 4 gigabytes. We implemented all changes in GraalWasm alongside all changes in Graal.js to support the WebAssembly JavaScript Interface.

Although this thesis improved GraalWasm in several areas, future work might further improve different performance aspects of GraalWasm and adopt new language features. Nevertheless, this thesis showed different ways to enhance an existing WebAssembly runtime.

Acknowledgment

I would like to thank Aleksandar Prokopec, Christian Wirth, Lukas Stadler, and Jan Stola, my technical supervisors on this thesis, for their constant feedback and for guiding me through all aspects of this thesis. Especially Aleksandar had a lot of valuable input and knowledge to share in our regular discussions. I would further like to thank my primary supervisor, Hanspeter Mössenböck, for his constructive feedback on my work. Last, I would like to thank all my colleagues, both at the university and the GraalVM team, as well as my friends and family for their constant support.

This thesis was performed in a research cooperation with, and supported by, Oracle Labs.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Bibliography

[1] Benedikt Spies and Markus Mock. "An Evaluation of WebAssembly in Non-Web Environments". In: *2021 XLVII Latin American Computing Conference (CLEI)*. 2021, pp. 1–10. DOI: 10.1109/CLEI53233.2021.9640153 (cit. on p. 1).

[2] Andreas Haas et al. "Bringing the Web up to Speed with WebAssembly". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: Association for Computing Machinery, 2017, pp. 185–200. ISBN: 9781450349888. DOI: 10.1145/3062341.3062363. URL: https://doi.org/10.1145/3062341.3062363 (cit. on pp. 1, 3, 4, 5, 6, 29).

[3] W3C. *WebAssembly JavaScript API*. Version 2.0. Apr. 19, 2022. URL: https://www.w3.org/TR/wasm-js-api-2/ (visited on 2022-09-07) (cit. on pp. 1, 10, 18, 56).

[4] Thomas Würthinger et al. "One VM to rule them all". In: *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. 2013, pp. 187–204 (cit. on pp. 1, 2, 12).

[5] Christian Humer et al. "A domain-specific language for building self-optimizing AST interpreters". In: *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*. 2014, pp. 123–132 (cit. on pp. 1, 2, 13).

[6] W3C. *WebAssembly Core Specification*. Version 2.0. Apr. 19, 2022. URL: https://www.w3.org/TR/wasm-core-2/ (visited on 2022-10-13) (cit. on pp. 3, 4, 6, 8, 21, 28, 32, 47, 50, 52, 55).

[7] Adam Hall and Umakishore Ramachandran. "An Execution Model for Serverless Functions at the Edge". In: IoTDI '19. Montreal, Quebec, Canada: Association for Computing Machinery, 2019, pp. 225–236. ISBN: 9781450362832. DOI: 10.1145/3302505.3310084. URL: https://doi.org/10.1145/3302505.3310084 (cit. on p. 4).

[8] Elliott Wen and Gerald Weber. “Wasmachine: Bring the Edge up to Speed with A WebAssembly OS”. In: *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. 2020, pp. 353–360. DOI: 10.1109/CLOUD49709.2020.00056 (cit. on p. 4).

[9] DWARF Debugging Information Format Committee. *DWARF Debugging Information Format, Version 4*. June 10, 2010. URL: https://dwarfstd.org/doc/DWARF4.pdf (visited on 2022-11-14) (cit. on pp. 8, 43).

[10] Ecma International. *ECMAScript 2022 Language Specification*. June 2022. URL: https://www.ecma-international.org/wp-content/uploads/ECMA-262_13th_edition_june_2022.pdf (visited on 2022-12-27) (cit. on p. 9).

[11] Li Liang et al. “Express supervision system based on NodeJS and MongoDB”. In: *2017 IEEE/ACIS 16th International Conference on Computer and Information Science (ICIS)*. 2017, pp. 607–612. DOI: 10.1109/ICIS.2017.7960064 (cit. on p. 11).

[12] Thomas Kotzmann et al. “Design of the Java HotSpot™ client compiler for Java 6”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 5.1 (2008), pp. 1–32 (cit. on pp. 12, 46).

[13] Erik Meijer and John Gough. “Technical overview of the common language runtime”. In: *language* 29.7 (2001) (cit. on pp. 12, 46).

[14] Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. “GraalSqueak: A Fast Smalltalk Bytecode Interpreter Written in an AST Interpreter Framework”. In: *Proceedings of the 13th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. ICOOOLPS ’18. Amsterdam, Netherlands: Association for Computing Machinery, 2018, pp. 30–35. ISBN: 9781450358040. DOI: 10.1145/3242947.3242948. URL: https://doi.org/10.1145/3242947.3242948 (cit. on pp. 13, 15, 59).

[15] David Leopoldseder et al. “Fast-Path Loop Unrolling of Non-Counted Loops to Enable Subsequent Compiler Optimizations”. In: *Proceedings of the 15th International Conference on Managed Languages & Runtimes*. ManLang ’18. Linz, Austria: Association for Computing Machinery, 2018. ISBN: 9781450364249. DOI: 10.1145/3237009.3237013. URL: https://doi.org/10.1145/3237009.3237013 (cit. on p. 15).

[16] Jonathan Sorenson. *An introduction to prime number sieves*. Tech. rep. University of Wisconsin-Madison Department of Computer Sciences, 1990 (cit. on p. 22).

[17] S.J. Fink and Feng Qian. "Design, implementation and evaluation of adaptive recompilation with on-stack replacement". In: *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* 2003, pp. 241–252. DOI: 10.1109/CGO.2003.1191549 (cit. on p. 30).

[18] Raphael Mosaner et al. "Supporting On-Stack Replacement in Unstructured Languages by Loop Reconstruction and Extraction". In: *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*. MPLR 2019. Athens, Greece: Association for Computing Machinery, 2019, pp. 1–13. ISBN: 9781450369770. DOI: 10.1145/3357390.3361030. URL: https://doi.org/10.1145/3357390.3361030 (cit. on p. 30).

[19] Michael Sannella et al. "Multi-way versus one-way constraints in user interfaces: Experience with the deltablue algorithm". In: *Software: Practice and Experience* 23.5 (1993), pp. 529–566. DOI: https://doi.org/10.1002/spe.4380230507. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380230507. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380230507 (cit. on p. 34).

[20] Bui Tuong Phong. "Illumination for Computer Generated Pictures". In: *Commun. ACM* 18.6 (June 1975), pp. 311–317. ISSN: 0001-0782. DOI: 10.1145/360825.360839. URL: https://doi.org/10.1145/360825.360839 (cit. on p. 34).

[21] Robert W. Scheifler. "An Analysis of Inline Substitution for a Structured Programming Language". In: *Commun. ACM* 20.9 (Sept. 1977), pp. 647–654. ISSN: 0001-0782. DOI: 10.1145/359810.359830. URL: https://doi.org/10.1145/359810.359830 (cit. on p. 40).

[22] Manuel Rigger et al. "Bringing Low-Level Languages to the JVM: Efficient Execution of LLVM IR on Truffle". In: *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages*. VMIL 2016. Amsterdam, Netherlands: Association for Computing Machinery, 2016, pp. 6–15. ISBN: 9781450346450. DOI: 10.1145/2998415.2998416. URL: https://doi.org/10.1145/2998415.2998416 (cit. on p. 59).

[23] Salim S. Salim, Andy Nisbet, and Mikel Luján. "TruffleWasm: A WebAssembly Interpreter on GraalVM". In: *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 88–100. ISBN: 9781450375542. DOI:

10.1145/3381052.3381325. URL: https://doi.org/10.1145/3381052.3381325 (cit. on p. 60).

[24] M. Šipek et al. "Next-generation Web Applications with WebAssembly and TruffleWasm". In: *2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO)*. 2021, pp. 1695–1700. DOI: 10.23919/MIPRO52101.2021.9596883 (cit. on p. 60).