Ahead-of-Time Compilation of WebAssembly using Eclipse OMR

by

Petar Jelenkovic

Bachelor of Science, Faculty of Organizational Sciences, University of Belgrade, 2017

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

Master of Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor(s): Kenneth B. Kent, PhD, Computer Science
Gerhard W. Dueck, PhD, Computer Science

Examining Board: Suprio Ray, PhD, Computer Science, Chair
Michael W. Fleming, PhD, Computer Science
S. A. Saleh, PhD, Electrical Engineering

This thesis is accepted by the
Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

August, 2020

© Petar Jelenkovic, 2020
Abstract

The variety of available computing machines limits the portability of programs. The primary hindrance is that programs are designed towards an interface espoused by a machine. However, other systems can run programs designed for a different interface using a virtual machine.

An approach for enhancing program portability is to design programs for a virtual machine with a simplified interface. For example, programs written in the C language can be compiled to the WebAssembly code format. WebAssembly was chosen for this research due to its simple syntax, static structure, recent interest from the research community and existing implementation with Eclipse OMR.

The development of language virtual machines often includes the implementation of an interpreter and a compiler. While interpreters primarily provide a sound implementation described by a language specification, compilers have the additional requirement of generating optimized machine code. This task can be facilitated using the Eclipse OMR toolkit for language runtime construction.

The relocation infrastructure and shared code cache are two features for ahead-of-time (AOT) compilation in Eclipse OMR and are currently in development. The research in the thesis shows an implementation of these two Eclipse OMR AOT compilation features in a language runtime. The WebAssembly AOT compiler, called Wabtaot, that is presented in this thesis leverages Eclipse OMR relocation infrastructure and shared code cache features. The comparison of Wabtaot with WebAssembly
runtimes implemented using other compiler technologies demonstrates that the implementation of a language runtime using Eclipse OMR AOT compiler framework is viable and its performance is competitive. Relative to Wasmjit-OMR, a Web-Assembly compiler implemented using Eclipse OMR just-in-time compiler features, Wabtaot significantly reduces execution time for repeated execution of WebAssembly modules.
Dedication

To my parents, Zlatija and Goran, and to my sister, Marina.
Acknowledgements

This research was conducted within the Centre for Advanced Studies—Atlantic, Faculty of Computer Science, University of New Brunswick. The author is grateful for the colleagues and facilities of CAS Atlantic in supporting this research. The author would like to acknowledge the funding support provided by the Atlantic Canada Opportunities Agency (ACOA) through the Atlantic Innovation Fund (AIF) program. Furthermore, I would also like to thank the New Brunswick Innovation Foundation for contributing to this research.

I am very grateful to my supervisors, professor Gerhard W. Dueck and professor Kenneth B. Kent, for their support and guidance during my studies. I would also like to thank Mark Thom and Georgiy Krylov for the help and collaboration in research at the CAS Atlantic lab.
Table of Contents

Abstract .................................................................................. ii

Dedication ............................................................................... iv

Acknowledgments ................................................................... v

Table of Contents ................................................................... vi

List of Figures .......................................................................... ix

1 Introduction ........................................................................... 1

2 Eclipse OMR and WebAssembly .............................................. 5
   2.1 Virtual Machines ............................................................. 5
   2.2 Eclipse OMR ................................................................. 9
       2.2.1 Symbols ................................................................. 11
       2.2.2 Code Generation ................................................... 12
   2.3 WebAssembly ............................................................... 14
       2.3.1 Instructions ......................................................... 15
           2.3.1.1 Memory instructions ................................. 15
           2.3.1.2 Numeric instructions .................................. 16
           2.3.1.3 Control flow instructions ...................... 16
           2.3.1.4 Functions .................................................. 17
       2.3.2 Interaction with the environment ......................... 17
5.2 Results .......................................................... 45
5.3 Summary ......................................................... 51

6 Conclusions and Future Work 52

6.1 Conclusions .................................................. 52
6.2 Contribution .................................................. 53
6.3 Future Work ................................................ 53

Bibliography 57

Vita
List of Figures

1.1 Compilation of C/C++ to WebAssembly and its use in WebAssembly runtimes, adapted from [15]. ..................................................... 2

2.1 Virtualizing software allows running software designed for a different ABI and creates a virtual machine, adapted from [29]. ......... 6

2.2 Compilation of bytecode to machine code with Eclipse OMR JitBuilder, adapted from [5]. ......................................................... 10

4.1 Wabtaot compilation process. .................................................. 33
4.2 Wabtaot and the Eclipse OMR AOT compiler modules. .............. 37

5.1 Second run performance of V8 and Wasmtime relative to Wabtaot for PolyBenchC benchmarks (ratio, lower is better). ............... 45
5.2 Second run performance of V8 and Wasmtime relative to Wabtaot for PolyBenchC benchmarks (ratio, lower is better). ............... 45
5.3 First run performance of Wasmjit-OMR relative to Wabtaot for PolyBenchC benchmarks (ratio, lower is better). ...................... 46
5.4 Second run maximum resident set size of Wasmtime and V8 relative to Wabtaot for PolyBenchC benchmarks (ratio, lower is better). .... 47
5.5 Second run maximum resident set size of Wasmtime and V8 relative to Wabtaot for PolyBenchC benchmarks (ratio, lower is better). .... 48
5.6 First run maximum resident set size of Wasmjit-OMR relative to Wabtaot for PolyBenchC benchmarks (ratio, lower is better). ........... 48
5.7 Second run performance of V8 and Wasmtime relative to Wabtaot for PolyBenchC benchmarks (ratio, lower is better) . . . . . . . . . . . . . 49
5.8 Second run performance of V8 and Wasmtime relative to Wabtaot for PolyBenchC benchmarks (ratio, lower is better) . . . . . . . . . . . . . 50
5.9 First run performance of Wasmjit-OMR relative to Wabtaot for PolyBenchC benchmarks (ratio, lower is better) . . . . . . . . . . . . . 50
Chapter 1

Introduction

Virtual machines allow computer programs designed for one type of software or hardware interface to be executed on a platform with a different interface. A common use of virtual machines is to run software designed for a certain kind of operating system or processor architecture on other systems with possibly very different architectures or system software [29]. However, developing a virtual machine for each distinct pair of desired architecture/system interface and available architecture/system interface, is a demanding task. Several solutions are used to overcome the challenge of designing virtual machines and attain the benefits of software portability.

The problem of providing virtual machines for many desired platforms can be overcome by only providing support for a single, well-defined interface. Then, all programs written in high-level languages and compiled for this interface can operate efficiently with this virtual machine. Considering that programs written in the C language are mostly compiled to machine code, the goal is to provide virtual machine code that can be easily implemented in software. For this research, we operate with a virtual machine for the WebAssembly code format [23]. It is a good fit for this goal as it was designed as a compilation target for C programs and for use in a platform-agnostic Web environment. WebAssembly specifies a set of instructions that operate
Figure 1.1: Compilation of C/C++ to WebAssembly and its use in WebAssembly runtimes, adapted from [15].

primarily on values located on the stack, load and store values to a linear memory or branch to other instructions. Any operating system services defined for the format are specified in the WebAssembly System Interface (WASI) [11], which is not part of the official specification. However, compilers Emscripten [7] and Clang [4] compile C programs to WebAssembly binaries that use WASI, and, thus, the virtual machine presented in this thesis, Wabtaot, provides support for it.

This WebAssembly virtual machine, Wabtaot, represents one approach to WebAssembly code execution, called ahead-of-time compilation. In general, virtual machines can execute code either by interpretation or compilation [29]. The project WABT [3], which represents the parser for Wabtaot, uses interpretation, while Wasmjit-OMR [2] expands on WABT by allowing just-in-time compilation to machine code for frequently executed WebAssembly code sequences. On the other hand, Wabtaot compiles all instructions from a WebAssembly binary to machine code, before the execution begins. In doing so, all machine code generated by Wabtaot is stored to static memory. On future execution of the WebAssembly binary, the pertaining machine code is automatically loaded, processed and executed. One of the
goals for this research is to compare the performance of WebAssembly execution in Wasmjit-OMR and the proposed solution in Wabtaot.

To use a WebAssembly runtime on different machines, the developers have to think about cross platform support. To achieve this, the designers of Wasmjit-OMR and Wabtaot virtual machines would need to create multiple, different compiler implementations and to understand the intricacies of each target architecture. However, Wabtaot and Wasmjit-OMR compilers are implemented with Eclipse OMR [21], a toolkit for language virtual machine construction. In particular, the Eclipse OMR compiler construction framework facilitates development to the point where compiler designers only need to define intermediate language constructs, which are automatically optimized and translated into machine code for one of the supported processor architectures and operating systems environments. During the development of Wabtaot with Eclipse OMR, the Eclipse OMR toolkit is modified to allow effective implementation of the WebAssembly AOT compiler.

The use of the Eclipse OMR relocation infrastructure and the shared cache module in implementation of a WebAssembly runtime has not been researched and evaluated with any standard benchmarks previously. The main objective of this research is to explore ahead-of-time compilation features in Eclipse OMR. The relocation infrastructure and shared code cache are two features that facilitate this research. The Eclipse OMR AOT compilation features are explored using the WebAssembly AOT compiler as a case study. The insights from the implementation of the WebAssembly AOT compiler contribute to the development of the Eclipse OMR ahead-of-time compilation technology. This facilitates the implementation of other language runtimes using the same technology.

The thesis’s second chapter describes the concept of the language virtual machine, and provides an overview of WebAssembly and Eclipse OMR. Following that is an overview of related WebAssembly runtimes. Next, the proposed design of the Web-
Assembly Eclipse OMR-based ahead-of-time compiler is detailed. The implementation of the design is presented in the following chapter, which also describes the modifications of the Eclipse OMR compiler and JitBuilder components. The evaluation of the implementation compares the performance of Wabtaot with related WebAssembly runtimes. The final chapter presents the conclusions from the evaluation and possible improvements to the ahead-of-time compiler.
Chapter 2

Eclipse OMR and WebAssembly

2.1 Virtual Machines

Modern computers are very complex. To combat the complexity, computer systems are separated into a hierarchy of levels of abstraction. Between two subsequent levels of abstraction are well-defined interfaces, implemented in different manners by the lower level [29]. For example, machine code is defined in terms of an instruction set that is implemented in hardware. The separation enables the design of the different abstraction levels in the computer system to be done independently, while the interfaces facilitate interoperability between them. However, two levels of abstraction are not able to interoperate if the higher level of abstraction is not designed for the interface provided by the lower level [29].

Virtualization defines the interface of one level of abstraction in terms of a different interface of the same level. With virtualization, machine code designed for one instruction set can be executed on hardware that uses another instruction set. Apart from the instruction set, computer programs must conform to the complete application binary interface (ABI), which represents the interface to the shared hardware resources maintained by the operating system [25]. Programs written in high-level
languages interact with the operating system via system calls that implement the application programming interface (API), that includes a set of libraries that provide specific functionalities. An instruction set and interface to the operating system implemented in software on a machine that provides a different ABI, constitute a virtual machine [29]. Therefore, virtualization of an ABI allows programs developed for some instruction sets and/or operating systems to execute on different machines (Figure 2.1).

![Diagram](image)

Figure 2.1: Virtualizing software allows running software designed for a different ABI and creates a virtual machine, adapted from [29].

The primary benefit of virtualization is the relaxation of constraints imposed by interfaces [29]. For example, the Wine virtual machine allows user programs developed for the Windows operating system to operate in a Linux-based system [16]. Another benefit is the enhanced security provided by using software in an execution environment managed by a virtual machine. The virtualization software essentially shields the hardware and the native system from malicious or erroneous user programs. Modern operating systems provide each running process an isolated, virtual execution environment that prevents one process from accessing another process's memory space, unless given the necessary permissions [29]. However, portability of
programs compiled to machine code can only be achieved on machines with different instruction sets by creating a translation of the source instruction set to every other instruction set. Considering the complexity of instruction sets, this can be a daunting task.

In order to enhance portability, some high-level languages are compiled to a virtual machine code. This type of machine code conforms to a virtual instruction set that is designed to be compact and easily translated by software to many different instruction sets supported by hardware. The virtual machine code, often called bytecode [20], is executed by a virtual machine that provides the API for using system-provided services. Two common modes of execution are interpretation and compilation [29]. With interpretation, each bytecode instruction is analyzed and multiple operations are performed to emulate the required behavior. However, each subsequent time the bytecode program is executed, the analysis and emulation must be done again. Compilation on the other hand translates the bytecode program or part of it to an equivalent program in machine code that executes on the available hardware, and caches the translated code for future use. The cached compiled programs can be executed without necessarily requiring a repeat of the analysis.

While compiled code executes faster than interpretation of its source bytecode [20], the design of virtual machines introduces certain particularities. First, analysis of the bytecode must be performed in addition to compilation, which overall could be a lengthier process than interpretation. This is an important trade-off for the time taken before the program begins execution, known as startup time. For example, a server application is expected to start responding to requests sooner if it begins by interpreting code, than if it is first compiled to machine code. Secondly, interpretation is often performed in tandem with compilation. In many high-level language virtual machine implementations, a program is interpreted and profiled simultaneously. The profiling records primarily which methods are the most executed. Based
on this data, the most executed methods are deemed hot, and are compiled to machine code at some point during interpretation, usually when a threshold number of separate executions of that method is reached. This is called just-in-time compilation, alluding to the fact that only the methods that are frequently executed are translated during the run of the program [29]. Any time in the future the program is run again, frequently executed methods will again be compiled when the threshold is reached.

While interpretation and just-in-time (JIT) compilation enhance portability [29], other benefits also accrue. During program execution in a high-level language virtual machine (HLL VM), profiling data is gathered about the current aspects of the execution that are influenced by current inputs. This allows the compiled code to be optimized for the current program inputs and environment, leading to better performance. Furthermore, security can be guaranteed across all operating systems and hardware where the VM is implemented [29]. There are also other features, like support for dynamic types or dynamic code loading, which might not be available in certain operating systems, but are always supported in certain HLL VM implementations.

After program execution ends in the JVM, just-in-time compiled code is discarded and the occupied memory is freed. Future executions of the program will require the reiteration of interpretation and JIT compilation. However, compiled code can also be cached and reused in future executions. In this way, some code that is generated ahead-of-time, can be executed without prior analysis. The caveat is the lack of optimizations that are usually more emphasized in JIT generated code, however the primary benefit is decreasing the startup time [31]. After more profiling data is available, some methods are recompiled to generate optimized code.

Compiled code requires the use of accurate memory addresses. For example, in machine code, a function is called by its location, indicated by the memory address.
In just-in-time-compilation, the addresses of these called functions are maintained during the current program execution. However, in ahead-of-time (AOT) compiled code, the addresses can differ between separate program runs. Therefore, relocation operations must set the addresses of all instructions that reference them [27]. While this operation adds further complexity and requires greater execution time from the VM, the performance improvements, primarily in startup time, should offset the costs.

Ahead-of-time compilation is useful for embedded environments. Considering that memory and processing resources are limited in embedded systems, using exclusively interpretation or JIT compilation can incur a significant performance penalty. It is shown that performance benefits of using AOT code is significant [32]. Furthermore, on some systems, dynamic compilation is not allowed. In this case, AOT must be used exclusively [24].

2.2 Eclipse OMR

A high-level language virtual machine is a complex system that most often is composed of an interpreter and a dynamic or static compiler. An interpreter designed in a high-level language can be recompiled or executed by a virtual machine on any platform that provides the necessary compiler or language virtual machine. On the other hand, the dynamic or static compiler of a virtual machine must generate efficient assembly code for all different platform ABIs it is intended to be used on. This requires specialist knowledge on the part of a HLL VM implementor on many different instruction sets and operating systems. Furthermore, some HLL VM specifications include garbage collection, meaning automated memory management done by the VM, which can be done in different ways for different languages and/or programs.
Figure 2.2: Compilation of bytecode to machine code with Eclipse OMR JitBuilder, adapted from [5].

In short, all this complexity requires a substantial effort to implement a language virtual machine. One way of dealing with the complexity is using a language runtime framework (runtime is a synonym for high-level language virtual machine). In the research presented in this thesis, the Eclipse OMR framework [21] is used to design and implement a static bytecode-to-machine code translator. Eclipse OMR [19] allows compiler development by providing facilities for defining bytecode functions as structured OMR internal language (OMR IL) trees, which are then automatically compiled and optimized for several different platforms (ABIs) (Figure 2.2). While the developers can construct OMR IL trees manually, the JitBuilder API [19] simplifies development by providing commands for construction of IL sub-trees, which represent a functionality like basic mathematical operations, local variable access
and various control flow management operations. Considering that some HLL VM implementations need to provide garbage collection, OMR provides a facility for ready-made garbage collection policies that language runtimes can opt to use.

When defining a bytecode compiler with the JitBuilder C++ API, several different classes must be used. Firstly, a user must provide a MethodBuilder subclass that will define the method name and signature and generate the OMR IL trees. From the perspective of the OMR compiler, the MethodBuilder is the interface to managing a method sent for compilation. A MethodBuilder usually uses the BytecodeBuilder to construct OMR IL for every bytecode defined for a language function. While the commands provided by the IlBuilder class generate sequences of IL commands for a certain operation, the BytecodeBuilder simplifies the management of control flow for developers. Commands from the IlBuilder operate on typed values that are located on a stack. Users can define their custom types using the TypeDictionary class, although basic types available in C are provided by Eclipse OMR. The typed values can be stored on a stack that can be completely implemented by a language virtual machine or it can be emulated with the Virtual Machine Operand Stack. The virtual stack provides the common stack operations, but also allows synchronization with another stack, possibly the one used by the interpreter.

2.2.1 Symbols

During OMR IL generation, defined functions and variables are referenced with symbols. Any operation declared in the OMR IL that uses variables or calls functions must include the related symbol reference. There are several types of symbols in the Eclipse OMR compiler, primarily distinguished by what they represent [19].

**Automatic** symbols represent local variables and their values will be stored on the stack in the generated machine code [17].
Parameter symbols represent function arguments and their values will be stored in registers, or on the stack, in the generated machine code [17].

Label symbols represent a location in code, in this case an instruction.

Static symbols represent variables that are stored in a memory location that is not represented by automatic or parameter symbols, nor allocated on an object heap [17].

Resolved Method symbol represents an already compiled function.

Symbols are registered in the MethodBuilder, often during initialization. At this time, static and resolved method symbols are registered with a memory address that will be referenced by generated machine code instructions. Later, when the symbols are used for an IlBuilder operation, for example, store and load operations for variables and call operations for functions, the actual symbols will be created from the registered data. Then, the created symbols of variables and functions will be used in the construction of OMR IL trees.

### 2.2.2 Code Generation

Eclipse OMR generates machine code based on the intermediate language trees defined by the user of the JitBuilder API [19]. The IL trees can also be defined directly and supplied to the code generator. Once the IL trees are defined, the automatic optimization infrastructure modifies the trees, primarily to reduce their size and create a streamlined OMR IL structure for automatic compilation. The optimized IL trees are further optimized for specific instruction sets. Eclipse OMR supports compilation of IL trees for Intel X86, AMD64, Z, PowerPC and ARM architectures. Each supported architecture is represented by a separate code generation module. The code generation is performed in multiple phases, notably instruction selection and binary encoding. During instruction selection, OMR IL tree nodes are evaluated
and translated into instructions for the pertaining architecture. The goal of this phase is to select the minimal number of instructions that use CPU cycles frugally. In this phase, placeholder registers are applied. Later, during the register assigning phase, optimal register allocation for the selected instructions will be attempted. Ultimately, during the binary encoding phase, selected instructions will be generated in memory. At this point, the generated instructions are allotted a memory location that can be used to register the required code relocations. Symbols that represent a memory location, like static and resolved method symbols, are associated with relocations. After the relocations are applied, the generated machine code can be executed by the user until the current JitBuilder instance is terminated.

The Eclipse OMR framework does not currently provide an integrated relocation or caching system required for an efficient implementation of an ahead-of-time compiler. However, using the work-in-progress relocation framework subsystem and the shared cache features, the ahead-of-time compiler envisaged for this research could be realized [27]. Firstly, the relocation framework in Eclipse OMR represents a modified version of the relocation infrastructure available in the Eclipse OpenJ9 codebase. This enables its design to be compatible with Eclipse OMR as it is derived from Eclipse OpenJ9. The design requires that developers of certain IL constructs define the necessary relocations which will be applied to the output machine code. Additionally, the developers using the Eclipse OMR relocation framework must also trigger the relocation process for each generated function and implement the logic for storing or performing the relocation. The relocation framework also does not make assumptions on the structure or type of storage used to cache the generated functions.

The storage used in this research is the shared cache, refactored and ported from Eclipse OpenJ9 [30]. Similarly to relocations, the developer of an Eclipse OMR-based language runtime must provide a dedicated shared cache structure that is
built upon the shared cache framework. However, the shared cache framework and the relocation framework can interoperate efficiently once the developers implement concrete solutions for their language runtime.

2.3 WebAssembly

In order to exercise the AOT compilation features of Eclipse OMR, the research presented in this thesis examines the performance of a WebAssembly runtime. WebAssembly is conceptually a stack-based virtual machine designed to be used in Web environments [23]. The origin of WebAssembly is rooted in the need for a fast and safe bytecode that will be the compilation target of other high-level languages, in particular C and C++. This way, HLL programs compiled to WebAssembly bytecode can be executed in Web browsers, while achieving performance similar to machine code executed by a physical processor. Furthermore, considering that WebAssembly programs are transmitted on the Web, safety is a fundamental concern. This implies that the design of WebAssembly cannot be identical to an assembly language for existing hardware.

The WebAssembly virtual machine executes instructions that operate on values located on the stack. All values have a 32-bit or 64-bit integer or floating-point type. Instructions operate on values located on the stack, or branch to other instructions. Apart from the stack, values can also be defined as global variables called globals, or be stored in a linear byte array called memory. There are specific instructions that access values stored in global variables and memories and push them onto the stack, as well as those that store values to globals and memories. Furthermore, the memory space, although limited, can be dynamically extended with a specific instruction; memory shrinking instructions are not currently defined in the specification.

Similarly to HLLs, WebAssembly specifies functions as a grouping of instructions.
Functions have parameters as values passed to the function, local variables, return values and, optionally, names. All functions belong to a module, which is a single WebAssembly file. When one function is calling another, the callee can be referenced by its name, or by its ordinal position in the module. Another way of calling functions is through the use of indirect calls. An indirect call is performed using an index into an array of functions denoted by their position in the module; this array is called a table. Functions defined in other modules can be called by first defining an import clause. Only functions that have been noted with export declarations from a module can be called. The import and export clauses also allow accessing globals and memories from other modules. Furthermore, the import declaration can be applied to the VM-specific values and functions that can be used by WebAssembly programs, while exports allow the runtime environment to access functions and values from a WebAssembly module.

### 2.3.1 Instructions

WebAssembly instructions largely operate on values located on the stack or push new values to the stack. Some instructions push values of variables on the stack: `get_local i` pushes the value of local variable `i` to the stack, while `get_global i` performs the same action for global `i`. The values of variables are set with `set_local i` for local variable `i` and `set_global i` for global variable `i`. These two instructions pop the value from the stack [23].

#### 2.3.1.1 Memory instructions

Memory values are loaded and pushed on the stack with the instruction `load a o`, while the instruction `store a o` stores the value from the top of the stack to memory. The memory address for loading the values from, or storing values to is determined from the 32-bit base address `a` stored on the stack and the 32-bit offset.
immediate \( o \). Before the memory access is performed, the address is checked to be within range of the memory size. Instruction \texttt{grow\_memory} increases the memory size, while \texttt{current\_memory} returns the current memory size. The memory size is indicated in number of pages, whereby a page is 64KiB in size. Finally, the endianness of values in memory is always little-endian, which requires endianness conversion on big-endian platforms [23].

2.3.1.2 Numeric instructions

Numeric instructions are typed instructions that operate on values located on the stack. They can be divided in the following way [23].

**Unary** uses the value from the stack top and pushes the result of the same type. Some examples are \texttt{abs} and \texttt{popcnt}.

**Binary** uses two values from the stack top and pushes one result of the same type:. This includes \texttt{add} instruction for adding two values from the stack top.

**Conversions** change the type of the value on the stack top.

**Comparisons** compare two values from the stack top and push one binary result.

**Test** compare the value from the stack top and push the binary result.

2.3.1.3 Control flow instructions

WebAssembly supports structured control flow features with blocks. Considering that blocks can be nested, a branch instruction \texttt{br \( i \)} indicates a jump to the end of the block at level \( i \) relative to the branch instruction. If the block is enclosed at the branch target level with a \texttt{loop} instruction, the jump targets the \texttt{loop} instruction. Optionally blocks can have labels, and in this case, the branch instruction can include the block label to indicate the jump target. There is also the conditional branch
\textbf{br_if} with a single target and the table branch \textbf{br_table} where one of the multiple jump targets are dynamically selected. When the end of the block is reached, all stack values pushed during the execution of this block and all its nested blocks are removed. The inclusion of structured control flow simplifies WebAssembly emulation and debugging [23].

\textbf{2.3.1.4 Functions}

Functions represent specific blocks which can take input values and produce a result value. The function’s signature can include the types of the input arguments, the return value and the types of local variables. The input values of the function are accessed like local variables, with instructions \texttt{get\_local} and \texttt{set\_local}. When the end of the function block is reached, the value on the top of the stack must match the expected return value. The jump to the end of the function block is signified with either a branch that targets the outermost block or the \texttt{return} instruction. Direct function calls are signified by the \texttt{call} instruction denoted by the name or ordinal number of the function in the module. Indirect function calls with \texttt{call\_indirect} are performed with an index into a global function table. An indirect call can be performed only if the index is within the value range of the table and the signature of the callee matches the expected signature. Once the call is finished, the stack height should be appropriate to the function signature and the execution proceeds with the next instruction in the sequence [23].

\textbf{2.3.2 Interaction with the environment}

Functions, memories, globals and tables can be shared with the environment via import and export declarations [8]. Any export defined by a unique name can be used by the embedding environment of the WebAssembly module. For example, this allows referring to WebAssembly functions in JavaScript code. On the other hand,
import declarations are denoted by the name of the imported function, memory, global or table preceded by the name of the module to which it belongs. The imports can be referenced by WebAssembly instruction in the same manner as functions, memory, globals or table defined in the module.

Imports and exports allow WebAssembly programs to interact with the operating system and use its services. Generally, WebAssembly code uses features provided by the surrounding JavaScript environment and browser through the Web API [12]. However, there is no standard API for WebAssembly in non-Web environments, in particular stand-alone WebAssembly runtimes, which is the topic of the research in this thesis. One such API is the WebAssembly System Interface (WASI), currently a work-in-progress [11]. WASI provides IO-related functionality available in POSIX libc. However, features not supported in WebAssembly, like multithreading and multitasking, are omitted. The primary focus in this research is on the WASI file IO functions (\texttt{fd\_write}, \texttt{fd\_seek}, \texttt{fd\_close}) and program argument functions (\texttt{args\_get}, \texttt{args\_size}).

### 2.3.3 Generating WebAssembly modules

Programs written in C or C++ can be compiled to WebAssembly using the Emscripten compiler [7]. By default, Emscripten outputs both a JavaScript driver file and a WebAssembly module that is run by it. However, it can also generate stand-alone WebAssembly modules that use the WASI API to access system services. Apart from the WASI-specified external functions, Emscripten describes additional external functions imported by WebAssembly modules. These additional functions are primarily used to communicate with a JavaScript execution environment, but have to be resolved even in stand-alone modules. For example, if the dynamic memory growth is used in the generated module, the stand-alone WebAssembly runtime running the module must implement the function \texttt{emscripten\_notify\_memory\_growth}. 

18
High-level language programs can also be compiled with the Clang compiler to stand-alone WebAssembly modules [4]. In the research presented in this thesis, Clang-generated module support was not implemented.

Ultimately, WebAssembly also provides a textual representation that enables manual creation of WebAssembly programs. The textual representation is designed to be human-readable as it is common for users to access the source of Web-based programs [23]. Before execution, textual representation of programs must be translated to the binary format.

### 2.3.4 WebAssembly Binary Toolkit

The WebAssembly Binary Toolkit (WABT) project primarily aims to provide a completely specification-compliant WebAssembly runtime implementation with an interpreter [3]. In addition, the project provides other tools useful for use of WebAssembly bytecode or text files, including translators between the two forms. An important part of WABT is the collection of tests for WebAssembly implementations, and their compliance with the specification.

The WABT interpreter parses and executes WebAssembly modules [3]. During the parsing phase, the input WebAssembly module is processed to initialize the runtime environment settings. This includes the initialization of the module’s memory and table, and defining their initial contents and size. Following is the generation of intermediate bytecode, which is similar to the WebAssembly binary format. However, all instructions in the module are parsed and analyzed, and an intermediate bytecode is generated to be interpreted. Of particular interest are branching instructions and function calls where accurate bytecode offsets must be determined before the start of interpretation. In this case, branch targets signified by labels in WebAssembly bytecodes are translated to offsets in the intermediate bytecode. Similarly, WebAssembly function calls are translated to represent the accurate callee index in the
intermediate representation bytecode. After the parsing phase, the intermediate bytecode is interpreted [3]. By default, only the labeled start function is run by the interpreter. On the other hand, all exported functions can also be run if the interpreter is configured to do so. The WABT interpreter implements all WebAssembly instructions, except control flow instructions, which are omitted from intermediate bytecode. Conversely, instructions not defined in the WebAssembly specification are added to the intermediate bytecode.

- **InterpCallHost** instruction represents a call to a host implemented function. The only host function in the WABT interpreter is a printing function, however it does not conform to the WASI specification.

- **InterpAlloca** is an instruction for allocating local variables on the stack at the start of the function.

- **InterpBrUnless** replaces the WebAssembly instruction `br_if` with a reverse condition.

- **InterpDropKeep** has two immediates that indicate the number of values to keep on the stack and the number to drop from the stack.

### 2.3.5 Wasmjit-OMR

Considering that WABT does not provide a WebAssembly-to-machine-code compiler, the Wasmjit-OMR open-source project provides a just-in-time compiler implemented with Eclipse OMR [2]. However, it does not support all WebAssembly features or instructions. Furthermore, after the runtime finishes the execution of a WebAssembly module and the module instance is destroyed, all compiled functions are discarded [18]. The just-in-time compiled functions access the interpreter stack when receiving
parameters from the interpreter or pushing the return value for the interpreter. The first just-in-time compilation occurs after engaging the interpreter. Further compilation can be triggered by function calls from already compiled functions. Nevertheless, it is a starting point for the WebAssembly AOT compiler researched in this thesis. The Wasmjit-OMR project adds a compiler for frequently executed functions in the interpreter. By design, the Wasmjit-OMR project extends the WABT interpreter program and execution of WebAssembly modules is done in the same way until a function call is interpreted. During a function call, if the function has already been compiled, the compiled function code is executed. Otherwise, the invocation threshold is checked, and if reached, the WebAssembly bytecode of the callee function is compiled to machine code. If the compilation fails, the function is interpreted.

2.3.5.1 Wasmjit-OMR Implementation details

The just-in-time compiler is implemented with the Eclipse OMR MethodBuilder components [2]. Every supported WebAssembly instruction is translated by a sequence of Eclipse OMR BytecodeBuilder commands. The process is very similar to that of the ahead-of-time compiler in Wabtaot, which inherited, extended or else modified the implementation of many instructions from the just-in-time compiler in Wasmjit-OMR.

Wasmjit-OMR introduces a new type in its implementation of TypeDictionary for representation of stack values. The new type Value is as large as an interpreter stack value, but it can contain a value of 32-bit or 64-bit size. A utility is introduced for conversion of C++ basic types to OMR IL types to facilitate the use of JitBuilder commands for OMR IL tree construction. This TypeDictionary implementation is used to initialize FunctionBuilder, the Wasmjit-OMR MethodBuilder subclass [2]. In the initialization, functions, other than the one being compiled, that will be referenced in the generated machine code of the function being compiled, are defined
as resolved method symbols (Chapter 2.2.1). The definition includes the name, signature and location of the function if it is available. Obviously, functions that have not yet been compiled cannot be identified by a memory address location. After the FunctionBuilder has been initialized, it is submitted to the Eclipse OMR compiler [2].

Eventually, the Eclipse OMR compiler will call the buildIL function in MethodBuilder subclasses to construct OMR IL tree representations of the entire bytecode function submitted for compilation. In Wasmjit-OMR, the buildIL function operates until all bytecodes in the function are translated into OMR IL. Each bytecode in the function being compiled is represented with a BytecodeBuilder object. A bytecodes program counter value in the WABT intermediate representation and BytecodeBuilder object are dynamically added to a list that is iterated upon in the buildIL function. Each list item is used to translate a WABT intermediate representation bytecode into OMR IL [2].

The notable details of translation of certain WebAssembly and WABT intermediate representation instructions will now be presented.

- Binary numeric instructions are implemented directly, with single IlBuilder commands for OMR IL tree construction. For example, the WebAssembly instruction add is implemented as the IlBuilder command Add.

- Unary numeric instruction sqrt is translated to a function call to the C standard library function SQRT. abs, neg and trunc instructions are implemented indirectly, with multiple IlBuilder commands. For example, neg is implemented as a multiplication of the operand with -1. Other unary numeric instructions are not implemented.

- Other numeric WebAssembly instructions, conversions, comparisons, and tests, are implemented with single IlBuilder commands.
• The instruction `br i` is implemented by connecting BytecodeBuilder objects. The value of instruction immediate `i` is read and the pertaining BytecodeBuilder object is searched in the list. If the BytecodeBuilder object is available in the list, it is immediately declared as the successor of the current BytecodeBuilder object. Otherwise, the BytecodeBuilder is created and stored in the list alongside its program counter value before being declared as the successor.

• The instruction `InterpBrUnless` is implemented similarly to the `br` instruction. However, the successor BytecodeBuilder is declared conditionally, based on the value on top of the stack.

After emitting OMR IL for each bytecode that does not involve branching, the BytecodeBuilder instance of the following bytecode is made the successor of the current BytecodeBuilder instance [2]. These instruction implementations are inherited in the Wabtaot compiler. However, there are modifications in the implementation of the ahead-of-time compiler in comparison to Wasmjit-OMR due to the following features of the JIT compiler.

• It does not implement all WebAssembly instructions. In particular, this includes `memory_grow`, `memory_size` and `br_table` [2].

• During the transition from interpretation to running compiled code, function arguments are passed on the interpreter stack. Accordingly, when the execution of compiled code finishes, the return value is passed on the interpreter stack. Therefore, JIT-compiled code has references to the interpreter stack. Otherwise, it uses a dedicated stack for local values, usually relegated to machine registers in compiled code [2].

• When storing a value in memory or loading a value from memory, the JIT-compiled code calls an address translation function to determine the exact location of the desired value in the implementation memory.
• During function calls from compiled code, the called function might not be compiled. In this case, the function is compiled before execution if the invocation threshold has been reached [18]. Otherwise, it is interpreted.

• After execution of a WebAssembly module finishes, all compiled code is discarded [18]. If the same WebAssembly module is run again with Wasmjit-OMR, any machine code will be compiled again.

The tool em-interp allows running WebAssembly modules generated by Emscripten in Wasmjit-OMR [6]. Em-interp is an extension of Wasmjit-OMR that facilitates implementation of functions imported in WebAssembly modules. The implementation must register the name of the module and the name of the function, and the actual host function as a C++ lambda function.

2.4 Summary

Software designed for one architecture or system interface can be executed on other platforms using a virtual machine. Programs written in high-level languages can be compiled to a machine code instruction set, which can then be executed by a virtual or physical machine. The other possible target for compilation can be a virtual instruction set, such as WebAssembly. However, the WebAssembly virtual machine must be implemented on the available platform to execute the WebAssembly virtual instruction set. Eclipse OMR provides tools that facilitate the construction of virtual machines for virtual instruction sets, also known as runtimes. Wasmjit-OMR is a WebAssembly runtime implemented using Eclipse OMR. In the next chapter, other WebAssembly runtimes, which are not implemented using Eclipse OMR, are presented.
Chapter 3

WebAssembly Implementations

WebAssembly was officially introduced in 2017, with support in Google Chrome, Mozilla Firefox, Apple Safari and Microsoft Edge browsers. All browser implementations access the WebAssembly code using HTML and a JavaScript runtime environment. By specification, WebAssembly modules do not have a function that is the starting point for execution. However, a complementary JavaScript code sequence is used to begin execution of a WebAssembly module and provides the necessary external functions. The Emscripten compiler can be used to compile C or C++ code to WebAssembly and produce either a JavaScript and a WebAssembly file or a stand-alone WebAssembly file that is not intended to be run using a JavaScript file. Some WebAssembly runtimes, including Wasmtime and Wasmer, can execute the stand-alone files and provide the necessary external functions directly.

3.1 V8 language runtime

The V8 language runtime is a WebAssembly and JavaScript VM implementation primarily developed by Google. WebAssembly modules can only be run by V8 using the Google Chrome web browser or the node.js runtime. HTML and JavaScript are necessary to run WebAssembly code in a web browser, while V8 can execute Web-
Assembly using only a JavaScript driver. When WebAssembly was first introduced, V8 performance was evaluated using the PolyBenchC benchmark suite compiled to WebAssembly and compared to execution of the same C code compiled to a native binary [23]. When emulating WebAssembly, V8 uses primarily ahead-of-time compilation with a baseline non-optimizing compiler to quickly generate machine code [1]. Later, frequently executed code sequences are further optimized. However, each time a WebAssembly module is emulated, the complementary JavaScript is also emulated, which adds performance penalties, considering the different nature of JavaScript.

3.2 Stand-alone WebAssembly runtimes

The potential of WebAssembly is the portability of the generated bytecode, paired with close-to-native execution performance. However, Web browser and node.js runtimes require the parsing and execution of the complementary JavaScript and even HTML code. In order to circumvent this, WebAssembly runtimes can directly execute specially formatted WebAssembly code. The exact format necessary to determine the start of execution of a WebAssembly module is prescribed by compilers that generate WebAssembly bytecodes for a HLL source. Currently, notable runtimes that execute stand-alone WebAssembly modules provide support for formats prescribed by Clang and Emscripten compilers.

3.2.1 Wasmtime

Wasmtime supports the format prescribed by the Clang compiler WebAssembly generator [14]. WASI-specified functions are supported. The native code generator for Wasmtime is Cranelift [14]. The compiled code can be stored and later loaded from a cache, if the optimization level matches the current settings.
3.2.2 Wasmer

Emscripten-generated stand-alone WebAssembly modules can be executed by Wasmer [13]. However, apart from the WASI-specified external functions, Emscripten describes additional external functions imported by WebAssembly modules. These additional functions are primarily used to communicate with a JavaScript execution environment, but have to be resolved even in stand-alone modules.

3.2.3 TruffleWasm

TruffleWasm is a stand-alone WebAssembly runtime that uses a JVM as the backend [28]. WebAssembly bytecode is interpreted and translated into abstract syntax trees by the Truffle framework. The generated AST is then executed by the GraalVM. This enables interoperability with other languages that are implemented with the Truffle framework. Both Emscripten and Clang compiled code are executable using TruffleWasm, where dedicated Emscripten external functions are imported from the complementary JavaScript module.

3.3 Summary

In the presented WebAssembly runtimes support, the execution of binaries generated by either Emscripten or Clang. However, the V8 runtime uses JavaScript code to provide the support for the external functions specified by WASI or the C-to-WebAssembly compiler. Wasmtime, on the other hand, provides the support for external functions in the runtime and does not execute JavaScript code. The goal of the research in this thesis is to explore the ahead-of-time compilation features in Eclipse OMR, using a WebAssembly runtime as a case study. Therefore, this WebAssembly AOT compiler runtime implemented using Eclipse OMR should be a stand-alone WebAssembly runtime with static compilation features, similar in design...
to Wasmtime. The distinguishing aspect is the implementation of the ahead-of-time compiler using Eclipse OMR. The description of the design and the implementation of this WebAssembly AOT compiler runtime using Eclipse OMR is presented in the following chapter.
Chapter 4

Design and Implementation of Wabtaot

The work in this thesis presents a compiler for the WebAssembly language implemented with Eclipse OMR and integrating its AOT features.

4.1 Wabtaot Design

The Wabtaot compiler is extended from the Wasmjit-OMR compiler. Modifications pertain to devising features for using the generated machine code without referring to the interpreter to achieve some functionality. Furthermore, the runtime system should be able to utilize the machine code of compiled functions generated in previous runs, in addition to the machine code generated for the current runtime instance. Finally, the ahead-of-time WebAssembly compiler should be able to execute machine code generated by Emscripten, primarily for executing the PolybenchC benchmarks [10] compiled to WebAssembly.
4.1.1 Independent execution

This section discusses the differences between existing Wasmjit-OMR and proposed Wabtaot runtime environments, the latter is equipped with an ahead-of-time compiler. Before compiling, Wasmjit-OMR utilizes the interpreter to engage the initial emulation. After a certain number of invocations of an interpreted WebAssembly function, the compiler is engaged to compile that function to machine code. Subsequently, when the threshold call count is reached in either the interpreter or compiled code, the called function will be compiled [18]. The proposed design of the ahead-of-time compiler is to immediately compile every function in the module and to never perform interpretation of WebAssembly bytecode.

The just-in-time compiler in Wasmjit-OMR generates machine code that uses helper functions to translate addresses when accessing linear memory [2]. In Wabtaot, the ahead-of-time compiled code should be able to perform load and store operations by directly accessing locations in linear memory. The expected benefit of this approach is a reduction in execution time of WebAssembly with Wabtaot compared to the performance of Wasmjit-OMR.

To enable this, the Eclipse OMR relocation infrastructure is used. By using relocations at link time, the addresses of entities can be resolved before the first execution, and at each execution accessed directly. This implies that relocations must be designed for function calls and accesses of memories and global values, in addition to any other relocations that stem from the characteristics of the Eclipse OMR code generator. Due to the extensive error checking and the dynamic nature of indirect calls, the problem should be resolved with the use of a helper function. However, this must not compromise the execution independence.
4.1.2 Reusing compiled functions

Any compiled function must be saved to a binary file on a persistent storage device from which it can be loaded when it is needed. This way, only functions that have not already been compiled or that have been modified should be compiled and stored. The binary storage should allow fast access to the function that is needed and should provide a way to store relocation information. Once a function is loaded, all the entities referred to by the code in the function should have their addresses resolved. In a case when a certain entity’s address cannot be resolved, the machine code should not be executed. By avoiding repeated compilation of WebAssembly functions to native machine code, execution speed up should occur on repeated execution of the same WebAssembly modules.

A fitting solution is the Eclipse OMR shared cache infrastructure. The requirements of the design can be fulfilled with a simple cache structure that stores compiled functions and their relocations. An example of this is the Wasm Composite Cache [30] which defines a header that identifies the function and its size, while the relocations are located immediately following the function code. Considering that the shared cache infrastructure is an integrated part of the Eclipse OMR framework, it should provide a quick access to code manipulation within the OMR technology.

4.1.3 Running Emscripten-generated WebAssembly

Considering that the related work on WebAssembly runtimes uses the PolybenchC benchmarks to execute Emscripten, the goal of the research is to evaluate and compare this design with other WebAssembly runtimes. To achieve this, Wabtaot must fulfill several requirements.

First, the just-in-time compiler does not translate all WebAssembly instructions to machine code. This means that the ahead-of-time compiler must support all WebAssembly instructions present in PolyBenchC WebAssembly modules. Secondly,
Emscripten-generated WebAssembly code refers to functions that are provided by the execution environment. These functions fall into two categories: Emscripten-provided functions and WASI functions. To be able to correctly compile WebAssembly and execute the generated machine code, Wabtaot must implement the required runtime-provided functions.

4.2 Implementation

Implementing the design for the WebAssembly ahead-of-time compiler requires creating a separate WebAssembly compiler based on the Eclipse OMR JitBuilder technology. In addition, to add the necessary functionality to Eclipse OMR, several modifications have been performed on the Eclipse OMR codebase.

4.2.1 Wabtaot

The ahead-of-time compiler operates differently compared to the just-in-time compiler. When a WebAssembly module is loaded, no interpretation is performed (Figure 4.1). First, the system checks if any function defined in the module is already stored in the cache. If the function has been stored, the compilation will not be performed and the function will be immediately loaded from the cache. Otherwise, the function is first compiled, then stored to the cache and subsequently loaded from the cache. In the following step, all relocations that must be performed for every function are applied. Finally, after relocation, the functions are ready for execution. Depending on the type of WebAssembly modules, different compiled functions are directly called by the Wabtaot runtime. In all cases, only the WebAssembly function exports should be called. For Emscripten, only a single exported function named _start is called, while for the included tests all exports should be run. After running the designated compiled WebAssembly functions, the Wabtaot runtime shuts down.
4.2.1.1 Wabtaot Implementation details

Unlike Wasmjit-OMR, Wabtaot does not represent an extension to the WABT interpreter program. Wabtaot is a separate program that leverages WABT interpreter functions and modules. The first phase is parsing the input WebAssembly module, which subsumes initializing the runtime environment settings, the content of the modules memory and table, and generating the WABT intermediate representation bytecode. During this phase, the external modules, including WASI and Emscripten modules, are registered. This enables assigning an index to the imported functions from the registered modules and using the index to reference the functions in WABT intermediate representation bytecode. Although WASI and Emscripten are not WebAssembly modules, all imported functions in WebAssembly must be declared with the source module name. The WABT parser used in Wabtaot explicitly checks whether each declared imported function belongs to the declared module, but also allows spontaneous registration of imported functions for registered modules. The imported function is assigned the index by registering it for a registered module.

After parsing is complete, the compilation process begins. In the first step, a separate instance of `AOTFunctionBuilder`, MethodBuilder subclass for the ahead-of-time compiler, is initialized for every function in the input WebAssembly module. The initialization is done using an instance of `AOTTypeDictionary`, which is a subclass...
of the Wasmjit-OMR TypeDictionary. During initialization, any function parameters are defined as parameter symbols and any global variables and memory are defined as static symbols (Section 2.2.1). Defining static symbols was not originally possible through the MethodBuilder and the ensuing modifications are presented in subsection 4.2.2.2. Finally, all other functions that can possibly be referenced in the generated code need to be defined as resolved method symbols. This also includes all functions in the input WebAssembly modules, as function calls are made directly in Wabtaot-compiled code.

In the next step, the shared code cache is accessed to load each function. If the function has been stored in the cache and loading was successful, compilation can be skipped. Otherwise, compilation is performed using the previously initialized AOTFunctionBuilder instance. During compilation, the Eclipse OMR compiler invokes the buildil function of the AOTFunctionBuilder to construct OMR IL trees. The AOTFunctionBuilder uses the Eclipse OMR VirtualMachineOperandStack to implement the WebAssembly value stack. In the first step of OMR IL tree construction, the stack is initialized to hold 64-bits, which is the size of the largest values stored on the stack. Following that, all function parameters are pushed to the stack. Similarly to Wasmjit-OMR, the buildil function finishes executing when all bytecodes in the function being compiled have been iterated upon. Equally, each bytecode is represented with a BytecodeBuilder object and connections between the objects represent the order of operations. A bytecodes program counter value in the WABT intermediate representation, the BytecodeBuilder object, the VirtualMachineOperandStack instance and the current stack count are dynamically added to a list that is iterated upon in the buildil function. Each list item is used to translate a WABT intermediate representation bytecode into OMR IL. Before OMR IL is emitted for each list item, the stack stored as part of the list item is set as the current stack. The notable details of translation of certain WebAssembly and WABT
intermediate representation instructions that are distinct from Wasmjit-OMR will now be presented.

- `get_global` and `set_global` are implemented as loading or storing the value of the variable represented by a static symbol.

- `call` and `InterpCallHost` are implemented using the `IlBuilder CALL` command, which takes the parameters popped from the stack and the name of the function as arguments. The function name must be previously defined as a resolved method symbol for the current `AOTFunctionBuilder` instance.

- The `InterpBrUnless` instruction implementation differs from the Wasmjit-OMR implementation in the need of preserving the stack. Because two possible control flow paths can be followed during execution, the stack has to be duplicated and preserved for emitting OMR IL for both target bytecodes. This is done by storing the stack instance and the stack count to the list that is iterated in the `buildil` function.

- Another type of branching, `br_table` is implemented using the `IlBuilder TABLESWITCH` command. First, each of the possible branching targets are initialized as cases and their BytecodeBuilder instances are labeled as possible successors to the current BytecodeBuilder instance. Then, an automatic symbol is set to hold the operand for determining the chosen case. Finally, the `TABLESWITCH` command is used with the list of initialized cases, the BytecodeBuilder instance for the default case and the operand symbol name. Similarly to two-way branching, for each possible case, a stack copy must be preserved.

- `store` implementation includes loading the value of a static symbol that represents the base location of the memory. The exact location is obtained by indexing the base location with the full address which is obtained as the sum
of the dynamic address from the stack and the static offset as the immediate to the bytecode. The value from the stack is then stored at the exact memory location. The implementation of load is done similarly, except that the value is loaded from the exact memory location.

- call_indirect is implemented using a helper function. The helper function is defined as a resolved method symbol. The command CALL is used with the helper function symbol name and three arguments that include the table index, the signature index and the entry index. The actual arguments for the target of the indirect call are stored to an array represented with a static symbol. In the helper function, the entry index is checked to be within the bounds of the table represented by the table index. Then, the function is found using the entry index and its signature is compared for equality with the supplied signature index. If all checks are successful, the call to the compiled function can be made. The current implementation of the helper function performs the call directly in C code and supports up to 6 parameters loaded from the array to be passed to the callee.

- memory_grow is implemented with a CALL command to a helper function based on the implementation from WABT. In Wabtaot, this helper function does not change the size of the allocated memory, but simply updates the data that tracks the expected number of pages allocated. The number of new pages to allocate is obtained from the stack, while the original number of pages is pushed onto the stack. Since the current design of Wabtaot does not allow relocations once compiled function execution begins, dynamic memory growth could not be supported with memory reallocation and direct memory access.

- popcnt is implemented with a CALL command to a WABT helper function.

Following the construction of OMR IL trees, the Eclipse OMR generates machine
code and relocations. Wabtaot stores the compilation output to the Wasm Shared Cache, which allows repeated compilation to be avoided in execution of that WebAssembly module in the future. After all functions have been compiled and stored to the cache, relocations are applied for each compiled function. Before that, the actual addresses of all relocatable symbols, in Wabtaot, primarily resolved method and static symbols, must be registered. For resolved method symbols, those are function addresses. For static symbols, those are addresses of the WABT implementations of globals and memories, or other variables. After applying relocations, the exported functions from the WebAssembly are run. To support execution of WebAssembly modules generated by Emscripten, only the exported function named _start is run. During execution of Emscripten-generated WebAssembly modules, functions defined by WASI might be called. Wabtaot implements three WASI functions that enable correct execution of WebAssembly programs evaluated in this thesis.

- \textit{fd\_write} outputs text, usually to the command line. This function has a pointer argument as an offset in WebAssembly memory where two consecutive 32-bit values indicate the offset to the actual text in WebAssembly memory and the size of the text to output. The Wabtaot implementation accesses the
text based on the indicated offsets and outputs the text of the indicated size to the system command line.

- \textit{args\_get} obtains program parameters passed from the system. The function has two offsets in WebAssembly memory as arguments, where the first indicates where the offset to the parameters should be stored, while the other indicates where the actual parameters in string format should be stored. If there are multiple parameters, the offset to each is stored as 32-bit values, starting from the location indicated by the first offset. In Wabtaot, the parameters passed on the command line are copied to the WebAssembly memory at the respective offsets.

- \textit{args\_size\_get} obtains the number and size of program parameters. The function has two offsets in WebAssembly memory as arguments, where the first indicates where the number of parameters and the other where the size of the parameters should be stored. In Wabtaot, the number and the total size of the parameters are stored at the indicated locations.

These functions were also implemented in em-interp (Section 2.3.5.1), to enable Wasmjit-OMR to execute Emscripten-generated WebAssembly modules. In the Evaluation chapter, the performance of Wabtaot and Wasmjit-OMR runtimes is compared. To facilitate this, the implementation of the three WASI functions in em-interp follows the implementation in Wabtaot.

The implementation of Wabtaot was, in part, extended from Wasmjit-OMR. However, the design envisaged for Wabtaot could not have been realized by directly saving and loading machine code generated by the just-in-time compiler. The primary reason is the lack of support for certain WebAssembly instructions. Other than that, the interpreting of WebAssembly bytecodes before compiling any machine code was deemed undesirable. The goal was to allow shorter execution time of WebAssembly
with Wabtaot on repeated execution of the same WebAssembly modules and thus avoid any interpretation on start up. The implementation that realized this design and was presented thus far uses Eclipse OMR features. However, to fully realize the envisaged design, some modifications had to be done to the Eclipse OMR codebase.

4.2.2 Eclipse OMR modifications

In the Eclipse OMR codebase some changes had to be made. First, for function calls, accesses of globals and memories and breaks on table new relocation types had to be introduced. Second, an interface for defining global variables had to be implemented. However, implementing these features allowed the system to operate soundly.

4.2.2.1 Implementation of Relocations

The relocation infrastructure in Eclipse OMR enables patching the code with appropriate addresses of relocatable entities. In WebAssembly, the instructions that necessitate relocations are function calls, accesses of memories and globals and table branch, a concept similar to a switch statement in other languages. Each call instruction in machine code generated by Eclipse OMR, as instructed by Wabtaot, uses the address of the called function. The accurate address of the called function can be determined at compile time if the called function has already been compiled and is located in memory. Otherwise, a placeholder address value should be used. In Wabtaot, the address of the callee is always considered to be unknown at compile time, thus a placeholder value that does not point to any other accessible memory location is used.

The only target platform for which the relocations are defined in Wabtaot is AMD64. The Eclipse OMR code generator outputs machine code, which uses an indirect call through a register. This way the relocation must be applied to the last instruction that sets the value of the register before the call is made with it. In the code gener-
ated by Eclipse OMR for the X86 architecture, that instruction immediately precedes
the call instruction in the generated code. In the Eclipse OMR X86BinaryEncoding
module, any relocations that ought to be applied to an instruction, must be registered
at the same time as the instruction is being generated. The MethodCallAddress re-
location type is introduced in Wabtaot for call instructions. It is, in part, ported from
the Eclipse OpenJ9 Java Virtual Machine implementation and its JIT-as-a-Service
feature. When this relocation is defined, the shortened name of the WebAssembly
function is recorded along with the offset from the beginning of the function to the
instruction it should be applied on. After the code is compiled and the relocations are
generated, the Eclipse OMR AOT infrastructure places them in the WASM Shared
Cache, a type of Eclipse OMR shared cache. Other relocations are also stored in the
cache in a similar fashion.

The accesses of memories and globals in WebAssembly are translated by the AOT-
FunctionBuilder and Eclipse OMR compiler to indirect accesses of memory values
stored in registers in X86 assembly. The relocations are applied to the last instruc-
tions that modify the value of the register used for indirect access before the access
is made. The relocation type is DataAddress, again in part, ported from the Eclipse
OpenJ9 JVM mainline codebase into Wabtaot.

Finally, the WebAssembly br_table instruction is implemented in AOTFunction-
Builder as a switch command. During compilation, the switch command from
MethodBuilder is turned into several jump instructions in X86 assembly. The jump
is made to an array of memory address values stored in memory locations preceding
the function call. The array contains as many addresses as there are locations in the
table defined in WebAssembly for the table branch instruction. The relocation type
BranchTable, introduced in Wabtaot, stores the offset from the location in the array
to the value it should hold. This way, when the relocation is applied to an address
in the array, the address is calculated from the designated location in the array.
4.2.2.2 Implementation of static symbol creation

Eclipse OMR JitBuilder API and MethodBuilder do not provide support for defining global variables. Memories and globals as defined by the WebAssembly specification were meant to be implemented as global variables in MethodBuilder. The modifications made to MethodBuilder and JitBuilder API (exposes MethodBuilder and others) during the research for the thesis enables defining global variables using MethodBuilder commands during the initialization of a MethodBuilder-derived class, in this case AOTFunctionBuilder. In MethodBuilder, the defined global variables are stored in a map, which is processed during referencing of a defined global variable. The referencing includes reading the value of a global variable or storing a new value. During the processing, for each defined global variable, a separate static symbol is created to represent it. In turn, the static symbols can then be referenced during code generation to emit a DataAddress relocation for it. The DefineGlobal command can also be applied for JIT-compiled code, as an address as well as a name can be defined during the creation of a global variable. The name is used for referencing the global variable in Eclipse OMR IL and IIBuilder commands.

4.3 Summary

Wabtaot is designed to reuse compiled functions and allow the execution of the compiled functions without falling back to the interpreter. Furthermore, the external functions required by the Emscripten-generated WebAssembly code are an integral part of the runtime. The implementation is based on Wasmjit-OMR and provides support for additional WebAssembly instructions. The use of the Eclipse OMR relocation infrastructure allows the change in implementation of some WebAssembly instructions in comparison to Wasmjit-OMR, by decreasing the use of helper functions. No interpretation is necessary, considering that all functions are compiled
ahead-of-time and stored to the shared code cache for later use. The implementation
necessitated modifications of the Eclipse OMR, primarily to realize the independent
execution aspect of the design. The final step of this research is the evaluation of
the implementation.
Chapter 5

Performance Evaluation

In this chapter, Wabtaot and its Eclipse OMR AOT backend is compared with other WebAssembly runtimes developed using different compiler infrastructures. The primary comparisons are with the V8 8.10 compiler [9] and Wasmtime 0.15 [14] for the PolyBenchC benchmark suite.

The results of the execution of PolyBenchC benchmarks with V8 were presented in the original WebAssembly specification [23]. Later work had also presented comparisons with the V8 runtime for PolyBenchC benchmarks [28][26][22]. Comparisons of Wasmtime and TruffleWasm for PolyBenchC benchmarks were also shown [28]. Considering the results presented in the related work, the Wabtaot runtime performance is compared to the performance of V8 and Wasmtime runtimes.

5.1 Evaluation Testbed

The setup for evaluation aims to measure the performance of WebAssembly runtimes when running modules generated by C-to-WebAssembly compilers. Each comparison is made by executing identical C code translated to WebAssembly using the Clang compiler for Wasmtime, or Emscripten for V8, Wabtaot and Wasmjit-OMR. The differences in interfaces that each runtime supports requires the use of different C-
to-WebAssembly compilers. For example, Wasmtime cannot execute WebAssembly modules generated by Emscripten, while the Wabtaot runtime exclusively supports interfacing used by Emscripten-generated bytecode. Furthermore, while V8 runtimes support Emscripten-generated code, they use a JavaScript file to engage execution of, and provide access to, system interfaces to WebAssembly code. For these reasons, Wasmtime executes WebAssembly generated by the Clang 11.0 compiler with the lowest (O0) optimization level and the stack size set at 8MiB. On the other hand, WebAssembly modules and the accompanying JavaScript files are generated by Emscripten 1.39.4 using the `-s ALLOW_MEMORY_GROWTH=1` flag and defining a file with the `.js` extension as output. Conversely, stand-alone WebAssembly modules are generated by Emscripten by defining a file with the `.wasm` extension as output.

PolyBenchC WebAssembly modules used for comparisons with Wasmjit-OMR are generated with `-DPOLYBENCH_DUMP_ARRAYS` flags set during compilation with Emscripten. The flag alters the C code to output the calculation results.

All runtimes are executed on an Intel i7-8700 platform with the Ubuntu 19.04 operating system and 32GiB of main memory. V8, Wasmtime and Wabtaot are run once before the measurement of execution is taken, primarily to eschew results from AOT compilation performed by Wabtaot or other runtimes. On the other hand, comparison with Wasmjit-OMR is based on the first run time, and measures the combined compilation and execution time during the first run for both Wabtaot and Wasmjit-OMR.

<table>
<thead>
<tr>
<th>Runtime</th>
<th>atax</th>
<th>biec</th>
<th>cholesky</th>
<th>ftdl-2d</th>
<th>gemver</th>
<th>gessumrv</th>
<th>mvt</th>
<th>seidel-2d</th>
<th>trisolv</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wasmjit-OMR</td>
<td>0.576</td>
<td>0.569</td>
<td>4.564</td>
<td>14.198</td>
<td>0.788</td>
<td>0.611</td>
<td>0.622</td>
<td>5.672</td>
<td>0.396</td>
</tr>
<tr>
<td>Wabtaot</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
</tr>
</tbody>
</table>

Table 5.1: First run execution time of Wasmjit-OMR relative to Wabtaot for PolyBenchC benchmarks (ratio, lower is better).
5.2 Results

Figures 5.1 and 5.2 show the relative performance of NodeJS V8 runtime normalized to Wabtaot execution time averaged over 10 runs. The data has been split into two
nine-member groups, based on the ranking of relative performance. Due to the design of the support for Emscripten-compiled code, the runtime needs to allocate over 2GiB for the module memory. During execution, frequent page faults are observed. For this reason, shorter-running benchmarks execute for more than 5 times longer than with the V8 runtime. However, longer running benchmarks which just as often access memory, tend to amortize the page faults over the entire running time and spend more time executing other instructions or generating output. Benchmarks 2mm and 3mm are close in execution time compared to V8, where the longest running 3mm is 6% slower than V8. All other benchmarks whose overall execution duration is longer than doitgen or ftdl-2d in both V8 and Wabtaot, are within 1.7× execution time in Wabtaot compared to V8, with three of them, 3mm, syr2k and syrk, being within 1.1×. The shortest running trisolv take 5 times as much as the V8 execution. However, Wasmtime is consistently faster in execution of PolyBenchC WebAssembly modules compared to V8 and Wabtaot.

In comparison with Wasmjit-OMR, Wabtaot performs up to 14× faster during the
compilation run of *fdtd-2d* (Table 5.1). The advantage of Wabtaot is lost during the compilation run of other short-running PolyBenchC benchmarks (Figure 5.3), where Wasmjit-OMR takes 45% of overall execution time of Wabtaot for compilation and execution of the shortest-running *trisolv*. On the other hand, during the second run when the entire WebAssembly module has already been compiled, the Wabtaot execution takes at most 11% of the time of the first run of the *trisolv* benchmark.

Figures 5.4 and 5.5 show the memory size of V8 and Wasmtime relative to Wabtaot while executing PolyBenchC benchmarks. Wabtaot allocates over 2 GiB for the module memory, which is extensively larger than the amount allocated even for the program with the most allocated memory, *gesummv*. For all benchmarks, the V8 memory footprint is consistently closer to Wabtaot than that of Wasmtime.

The relative memory size of Wasmjit-OMR in comparison to Wabtaot in Figure 5.6 is larger on average than V8 and Wasmtime. Here, Wasmjit-OMR maximum resident set size is at most 17% of that of Wabtaot.

---

**Figure 5.4:** Second run maximum resident set size of Wasmtime and V8 relative to Wabtaot for PolyBenchC benchmarks (ratio, lower is better).
Figure 5.5: Second run maximum resident set size of Wasmtime and V8 relative to Wabtaot for PolyBenchC benchmarks (ratio, lower is better).

Figure 5.6: First run maximum resident set size of Wasmjit-OMR relative to Wabtaot for PolyBenchC benchmarks (ratio, lower is better).
The division of long-running and short-running benchmarks is made on \textit{fdtd-2d} and \textit{doitgen}. While \textit{fdtd-2d} executes faster than \textit{doitgen} with V8, the fact that \textit{fdtd-2d} includes more abundant memory operations is detrimental to the design of Wabtaot’s implementation of memory manipulation. We have observed throughout the benchmark the issue of spending considerable execution time in system operations for virtual memory management. This should inform the future development of Wabtaot, which could enable performance improvements for short-running WebAssembly modules.

![Second run performance of V8 and Wasmtime relative to Wabtaot for PolyBenchC benchmarks (ratio, lower is better).](image)

The original implementation of WebAssembly memory in Wabtaot was inherited from the Wabt interpreter codebase [3]. The memory in the Wabt interpreter could actually be dynamically grown. However, in Wabtaot, WebAssembly memory is allocated once and does not subsequently resize. After observing the abundant page faults occurring during execution of benchmarks with Wabtaot, the implementation of memory was changed from using an initialized C++ standard library vector to using a standard library array. The implementation of WebAssembly memory was
changed this way to reduce the number of page faults during execution of PolyBenchC benchmarks with Wabtaot.

The results of the implementation change are shown in Figures 5.7 and 5.8. Now, V8
performs better than Wabtaot in only three long-running benchmarks, correlation, covariance and symm. For all other long-running benchmarks, Wabtaot performs better than V8. For example, the relative execution time of doitgen with V8 is now over 1.2× longer, while it was originally 10% shorter. Much shorter time is observed for execution of short-running benchmarks with Wabtaot. For example, cholesky and fdtd-2d execution time with V8 is more than 80% of the Wabtaot result. Furthermore, almost all other benchmark results are between 40%-60% range of relative execution time, while originally, these benchmark results were below 40%. Finally, the worst performing trisolv benchmark’s execution time more than halved. The difference in relative execution time for the new implementation in comparison to Wasmjit-OMR has only slightly improved (Figure 5.9). The new implementation affects primarily the execution of the compiled code, while compilation is less affected. Nevertheless, the largest relative compiled code execution time is gemver again, but has decreased from 7% to 3.8% of the overall time of the first run.

5.3 Summary

The performance of Wabtaot improves when the linear memory implementation is not initialized. In turn, the execution of many long-running PolyBenchC benchmarks is quicker with Wabtaot then with the V8 runtime. While none of the shorter-running benchmarks performance with Wabtaot is better then with V8, improvements in execution time are observable throughout. The performance of PolyBenchC benchmarks with Wabtaot is never superior to the performance of Wasmtime. Based on the evaluation results, further steps in the development of the WebAssembly ahead-of-time compiler with Eclipse OMR are presented in the following chapter, alongside the conclusions and contribution of the research in the thesis.
Chapter 6

Conclusions and Future Work

6.1 Conclusions

This thesis presented the use of the Eclipse OMR toolkit in designing and implementing an ahead-of-time WebAssembly runtime called Wabtaot. The design required that Wabtaot compiles every function in a WebAssembly module and store the results in persistent memory for future use. If the module’s functions have already been compiled, the stored compiled code would be loaded and executed. Machine code relocations and the use of persistent storage were the key features for enabling this design.

The design was implemented by using the existing or adding new features to the Eclipse OMR compiler construction toolkit. The compiler was inherited, modified and extended from Wasmjit-OMR to support ahead-of-time compilation of Emscripten-generated WebAssembly modules. This way, new instructions were implemented. To achieve effective reuse of compiled code, relocations were defined for previously-compiled code loaded from persistent storage.

The Wabtaot compiler provided support for execution of benchmarks in the PolyBenchC suite. To achieve this, support was provided for features from the WASI
specification and Emscripten-related features. For evaluation of the implementa-
tion, comparisons were made with V8, Wasmtime and Wasmjit-OMR WebAssembly
runtimes.

6.2 Contribution

The Eclipse OMR ahead-of-time compilation features are evaluated as part of the
WebAssembly runtime. In particular, the Eclipse OMR relocation infrastructure
and shared code cache modules are used to implement a WebAssembly runtime,
whose performance is compared to other WebAssembly runtimes implemented using
different technologies or the Eclipse OMR just-in-time compilation features. The
implementation of the WebAssembly ahead-of-time compiler produced modifications
of the Eclipse OMR API and presented a tangible use-case for the Eclipse OMR
relocation infrastructure and shared code cache.

6.3 Future Work

The results from the evaluation show that the performance of Wabtaot is comparable
with the V8 runtime for certain benchmarks. However, the relative execution speed
results are inconsistent with regards to individual benchmarks from the PolyBenchC
suite. In particular, the slowdown is the most evident with short-running bench-
marks. Therefore, the first steps in improving the Wabtaot runtime should address
this issue and provide an improved solution for these PolyBenchC programs. Wab-
taot should also provide support for WebAssembly modules produced by the Clang
compiler, which would enable a more equal comparison with Wasmtime. Finally, the
changes to the Eclipse OMR codebase and the insights from the implementation of
this ahead-of-time compiler can guide the development of other language runtimes
using Eclipse OMR AOT compilation technology.
Bibliography


ternational Conference on Virtual Execution Environments (New York, NY, USA), VEE ’20, Association for Computing Machinery, 2020, p. 88–100.


Vita

Candidate’s full name: Petar Jelenkovic
University attended (with dates and degrees obtained): Bachelor of Science in Information Systems and Technologies, University of Belgrade, 2017
Publications:


Conference Presentations: