ELF-based Code Storage Support for the Eclipse OMR Ahead-of-Time Compiler: A WebAssembly Use Case

by

Damian Diago D'monte

Bachelor of Engineering (Computer), Fr. Conceicao Rodrigues College of Engineering, University of Mumbai, 2016

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):	Gerhard W. Dueck, Ph.D., Computer Science
	Kenneth B. Kent, Ph.D., Computer Science
Examining Board:	Eric Aubanel, Ph.D., Computer Science, Chair
	Suprio Ray, Ph.D., Computer Science
	Richard Tervo, Ph.D., Electrical and Computer Engineering

This thesis is accepted by the
Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

August, 2021

© Damian Diago D'monte, 2021

Abstract

Over the years, Ahead-of-Time (AOT) compilation has drawn significant attention in the research community due to its ability to accelerate the startup of a runtime system. AOT compilation involves compiling, persisting and re-using existing compiled code in later runs, thereby avoiding costly re-compilation. Typically, a program is compiled and translated into machine language or native code, which persists in a binary container format. At present, the most commonly used code container options are either cache-based or object-based. Eclipse OMR is a collection of components, which are used to build robust language runtimes. The WebAssembly AOT compiler (*wabtaot*) is constructed using the Eclipse OMR JitBuilder library. Currently, the *wabtaot* compiled code is stored in a prototype of the Eclipse OMR shared cache. The goal of this research is to find a better lightweight AOT code storage option for resource-constrained systems. To achieve this goal, the Eclipse OMR ELF generation module is enhanced by implementing an ELF shared object generator that can store AOT data and use it as part of the Eclipse OMR AOT component. The design and implementation of the ELF shared object for Eclipse OMR is discussed and demonstrated using *wabtaot*. Evaluation is carried out by comparing the ELF shared-object approach with the existing shared cache in the *wabtaot* environment on metrics like execution speed, memory footprint, file size and sharing. By doing so, the execution time trade-offs, lower memory consumption and compact-size ELF objects are witnessed, thereby

indicating a possibility of using the lightweight ELF shared objects in resource-constrained systems.

Dedication

To my loving parents, Shobhana Diago D'monte and Diago Joseph D'monte, and my brother Samron.

Acknowledgements

I would like to express my deep and sincere gratitude to my supervisors, Dr. Kenneth B. Kent and Dr. Gerhard W. Dueck, for giving me this opportunity to do research and work under them. Also, I would like to thank them for their continual guidance and support.

I thank my fellow labmate and friend, Georgiy Krylov, for his invaluable help and guidance that let me through my masters journey. He has been a patient mentor, who always encouraged me to do better.

I am extremely grateful to our project manager, Stephen MacKay, who reviewed and provided feedback on almost every document, including research papers, proposal, and this thesis. I admire his excellent English writing/editing skills, and thank him for improving my writing skills. Thank you for making my thesis look better.

To IBMers, Younes Manton and Daryl Maier, thank you for all your support. I commend your in-depth knowledge in compilers and feel fortunate to have worked with you guys.

I thank research assistants, DeVerne Jones and Aaron Graham for providing system related support.

I 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.

Table of Contents

Abstract **ii**

Dedication **iv**

Acknowledgments **v**

Table of Contents **vi**

List of Figures **ix**

List of Tables **x**

Abbreviations **xi**

1 Introduction **1**

2 Background **4**

2.1 Executable and Linkable Format (ELF) 4

2.1.1 Sections and Segments 6

2.1.2 Dynamic Linking and Loading 9

2.1.3 Attributes of the ELF Specification 12

2.1.3.1 Broad Platform Support 12

2.1.3.2 Extensible and Flexible 12

2.1.3.3 Code Sharing and Reuse 12

2.1.3.4 Position Independent Code (PIC) 13

2.1.3.5 Dynamic Behavior 13
2.1.3.6 Tooling Support 14
2.2 Eclipse OMR 15
2.2.1 AOT Component 15
2.2.2 Relocations Infrastructure 17
2.2.3 Shared Cache 17
2.2.4 Extensible Classes 19
2.2.5 ELF Generation Module 20
2.3 WebAssembly 21
2.3.1 WABT 22
2.3.2 wasmjit-omr 22
2.3.3 wabtaot 22

3 Review of Runtimes **24**
3.1 Mono Runtime 25
3.2 Oracle HotSpot 26
3.3 Android Runtime 27
3.4 Eclipse OpenJ9 28
3.5 Zing Runtime 29
3.6 V8 29
3.7 Wasmer 1.0 31
3.8 Wasmtime 31
3.9 Lucet 32
3.10 Summary 33

4 Design and Implementation of Shared Objects **34**
4.1 Variability in ELF Generation Module 35
4.2 The aotelf Project 37

4.2.1 Extended AOT Component 38
4.2.2 Extensible Classes + Multiple Inheritance 39
4.2.3 Compilation in wabtaot 40
4.2.4 AOT Data Serialization 42
4.2.5 AOT Shared Object Composition 43
4.3 Wabtaot Modes and Shared Library Approach 45
4.4 Summary 47

5 Results & Evaluation 48
5.1 Experimental Setup and Benchmarks 48
5.2 Execution Time 49
5.3 Memory Usage 53
5.4 File Size and I/O Performance 55
5.5 Sharing and Concurrency 56

6 Conclusion and Future Work 58

Vita

List of Figures

2.1 View Perspective of ELF Objects [16] 5

2.2 ELF Shared Object Structure 8

4.1 Design of AOT-ELF Shared Object Generation within *wabtaot* runtime with Eclipse OMR. 35

4.2 The aotelf Compiler Project Directory Structure using Extensible Classes 37

4.3 Depiction of Serialization into ELF Shared Object using `AOTMethodHeader` 43

4.4 Structural Composition of an AOT Shared Object 45

4.5 wabtaot Shared Library Approach 46

5.1 Physical Memory Consumption of PolyBenchC Modules 53

5.2 Virtual Memory Consumption of WebAssembly Modules 54

List of Tables

3.1 Runtime Systems and their AOT features. 25

5.1 Comparison of the shared cache and the shared object execution time in **COMPILE RUNS** (in seconds). 50

5.2 Comparison of the shared cache and the ELF shared object execution time in **LOAD RUNS** (in seconds). 51

5.3 Comparison of the shared cache and the ELF shared "library" execution time in **LOAD RUNS** (in seconds). 52

5.4 Comparison of the shared cache and the ELF shared object execution time for 10 CONCURRENT **LOAD RUNS** (in seconds). 57

List of Symbols, Nomenclature or Abbreviations

AOT	Ahead-of-time
API	Application Programming Interface
ART	Android Runtime
ASLR	Address Space Layout Randomization
COFF	Common Object File Format
CRTP	Curious Recurring Template Pattern
DVM	Dalvik Virtual Machine
ELF	Executable and Linkable Format
GCC	GNU Compiler Collection
GOT	Global Offset Table
IoT	Internet of Things
JDK	Java Development Kit
JIT	Just-in-time
JVM	Java Virtual Machine
LLVM	Low-Level Virtual Machine
PHT	Program Header Table
PIC	Position Independent Code
PLT	Procedure Linkage Table
RSS	Maximum Resident Set Size
SCC	Shared Classes Cache
TIS	Tool Interface Standards Committee
TR	Testarossa
VM	Virtual Machines
WABT	WebAssembly Binary Toolkit
WASM	WebAssembly
WAT	WebAssembly text format

Chapter 1

Introduction

Computer programming languages such as Java, Python, and Rust require a managed runtime environment for program execution. Runtime environments provide automatic memory management, code compilation, execution and storage mechanisms. Language runtime environments use interpreters to execute instructions one-by-one directly, or a compiler is used to convert a source program into native code. A compiler generally takes the computer program code as input, translates it into machine code and stores the output in an object file or some other form. In computer systems, object files are binary representations of compiled programs storing code, data and control information [54].

Originally, Java runtimes only interpreted the bytecode line-by-line and executed it at runtime [6]. The problem with using only interpreters is the overhead of fetch-decode operations from the interpreting process. For instance, a piece of code in a loop needs to be retranslated every time, causing performance issues. To avoid this, Just-in-Time (JIT) compilers were introduced [6]. JIT compilers incur an one-time cost of compiling a block of interpreted code into native code and allowing for later reuse within the same instance of a runtime environment, thereby saving the execution overhead caused by an interpreter. Moreover, JIT compilation allows

compile-time optimizations. JIT compilation has enabled Java virtual machines to dynamically convert bytecodes into native code.

Another traditional compilation method is Ahead-of-Time (AOT) compilation, which converts the program code into machine (native) code and persists it in a code container [6]. The native code persisted in a code container can be loaded and re-used in later runs to directly execute it, thereby avoiding the overhead caused by the interpretation [6]. Therefore, reusing AOT compiled code rather than recompiling the same code again can enhance the startup time of the VM. AOT generated code needs to be stored in a container format such as an object format or equivalent. For example, C generates an object file with a *.o* extension, whereas Java generates a *.class* file with bytecode post compilation. Choosing an appropriate code container for an AOT compiler is crucial, because different code containers may have implications on runtime performance. These implications can affect the start-up time, memory consumption and file system utilization of the runtime that uses the code container.

A standard format that allows storing the binary code is the Executable and Linkable Format (ELF) [16]. ELF objects are widely used as code containers in Unix/Unix-like systems and have extensive tooling support. ELF objects are also used in resource-constrained environments, like embedded devices and IoT systems. ELF objects store the compiled code and data as well as holding control information that is helpful for execution. Eclipse OMR is a collection of language-agnostic components that can be used to build robust language runtimes. Its components include a garbage collection framework, JitBuilder and a threading library. The WebAssembly AOT compiler (*wabtaot*) is developed using the Eclipse OMR technology [45, 30, 42]. Currently, the wabtaot compiler uses the Eclipse OMR shared cache in order to persist its code and data [45, 42]. The shared cache is a fixed-size storage container, which is meant to store AOT data of all files that are compiled. If the shared cache

size is 300 MB and the compiled code size is only 50KB, then the shared cache will still consume 300 MB of memory. Also, the current version of shared cache does not support concurrent execution of programs.

This thesis presents a portable and lightweight code container option in Eclipse OMR for storing AOT code and data. The objective of this research is to enhance the AOT component of Eclipse OMR and provide an alternative code container option to store AOT code and data. Currently, Eclipse OMR can generate executable and relocatable ELF objects [54, 32]. However, these objects can neither share the code, nor perform operations like dynamic loading, which are desirable features [22, 21]. This limits the use of ELF objects as-is for AOT storing code and data. Therefore, the ELF generation module is enhanced and the *aotelf* project is implemented, which has the functionality to populate and write the AOT code and data into an ELF shared object. As wabtaot consumes Eclipse OMR, wabtaot is used to demonstrate the usefulness of the new container, i.e., the ELF shared object. The design choices adopted in this research enable the language runtime developers to extend the aotelf project and use it (or modify it) based on the their requirements.

The remainder of this thesis is structured as follows: Chapter 2 describes the background details of this research; Chapter 3 reviews various runtimes and their AOT support; Chapter 4 elaborates the design and implementation of the ELF shared object generator; Chapter 5 offers a preliminary performance evaluation that compares the ELF shared object approach with the existing shared cache; followed by Chapter 6 concluding this thesis with future work.

Chapter 2

Background

This chapter describes the specifics of an ELF object, the Eclipse OMR framework, including Eclipse components used in this work, and the WebAssembly format, including WebAssembly runtimes used in this research.

2.1 Executable and Linkable Format (ELF)

Executable and Linkable Format, also known as ELF, is a portable object file format developed by UNIX System Laboratories [54]. An ELF object file is generated as a result of the compilation process. Typically, an ELF object file contains compiled code, metadata, relocations and other data. The ELF header has a fixed position and is always placed at the start of the ELF object. It contains information like ELF type, architecture, version and length of section and program headers. The program header contains information about the segments, and the section header stores information about each section.

An ELF object presents dual views of its file contents [56]. As seen in Figure 2.1, these views are, namely, the linking view and the execution view. The linking view treats an ELF object as a set of sections described by a section header table, whereas the execution view treats the file as a set of segments described by a program header

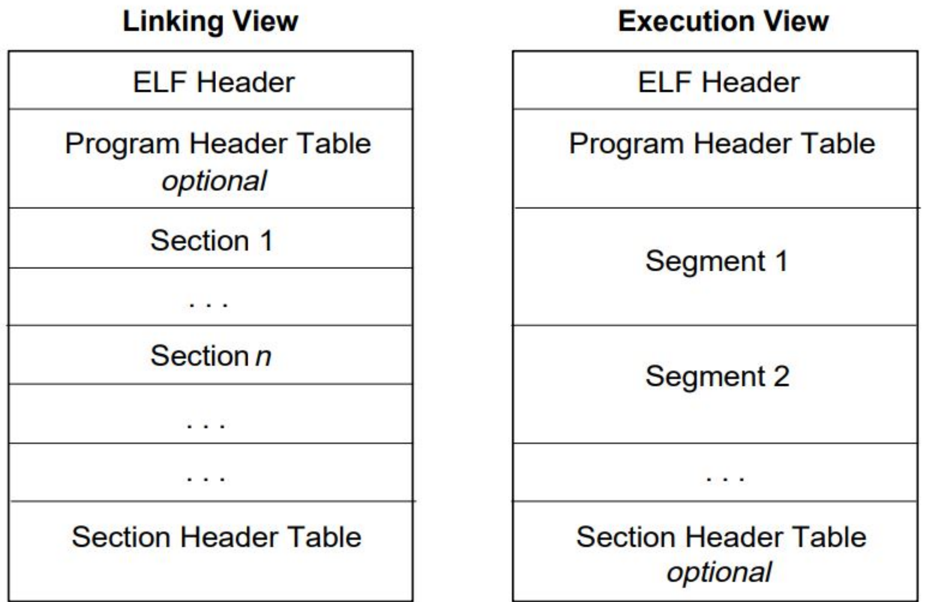


Figure 2.1: View Perspective of ELF Objects [16]

table. The linking view is from the compilers, assemblers and linkers perspective, while the execution view is from the loaders perspective [16].

The commonly used ELF object files are of following types:

- Executables: The executable object file contains execution-ready program code.

- Relocatable Object(*.o*): The relocatable object file contains code that is used for linking with other object files. Relocatable object files need to be processed by the linker before execution.

- Shared Object (*.so*): The shared object file (*.so*) is the most relevant ELF type to this research [54]. One of the features of the shared object is its dynamic behavior. This means that dynamic shared objects can be linked and loaded at runtime [16, 46]. Static object files are meant to be integrated in the final executable. Therefore, when a static object is modified, its executable also needs to be regenerated. If a dynamic shared object is used, the executable

only needs the path of the dynamic shared object. At runtime, the dynamic linker and loader, links and loads the code of the dynamic shared object, when it is requested by an executable. This allows shared objects to be modified or recompiled independently. The process of creating a shared library directly from a C program using the GNU Compiler Collection (GCC) is described below.

```
gcc -Wall -Werror -fPIC test.c
gcc -shared -o libtest.so test.o
```

GCC provides built-in support to produce an ELF object file as an outcome of the compilation activity. In the first command, GCC compiles the program `test.c` and stores the compiled code in the `test.o` file i.e., a relocatable object. The `-fPIC` flag indicates position independent code (PIC), whereas the `-shared` flag indicates that the library can be shared. The second command will create a shared object file of the relocatable object file. This generated `libtest.so` shared library can be linked to other executables or loaded at runtime.

2.1.1 Sections and Segments

An ELF object file is made up of sections and segments. However, there is a conceptual difference between sections and segments. Sections provide information about how information is organized in the ELF object [46]. Sections concentrate on the linking view and hold information such as instructions, data, symbol table and relocation information. A brief description of each section follows [63]:

- *.text*: Stores the compiled code (binary). This section usually has readable and executable permissions.

- *.data*: Stores all the initialized data that can be modified at runtime.
- *.rodata*: Stores data that is meant to be read-only.
- *.rela.**: Holds tabular information associated with relocation patches for *.text, .data and .rodata* sections respectively.
- *.symtab*: Stores all the symbols.
- *.dynsym*: Stores the symbols that are used in dynamic linking and loading.
- *.shstrtab*: Stores the object file's section header table that lets one locate all the file's sections. The dynamic symbol table is a sub-table of the symbol table.
- *.dynstr*: Stores an array that contains all symbol name strings.
- *.hash*: Holds the hash table, which provides fast access to symbol table entries without using a linear search.
- *.dynamic*: Holds the address and size of the string table, symbol table, hash table and relocation tables.
- *.got*: Holds the global offset table (GOT). GOT stores all the imported and exported global symbols of the file.
- *.plt*: Holds a procedure linkage table (PLT), which contains an entry for each non-local routine called from the shared library.
- *.got.plt*: *.got* runtime-writable section entries for lazy binding are placed in *.got.plt*. All other read-only entries are placed in the .got section

Similarly, in the execution view, a program header table containing multiple program headers describe to the program loader how a process image should be loaded in memory from the ELF object. If the ELF object is dynamically linked, the dynamic linker is concerned with dynamic linking and loading. The files that are used to

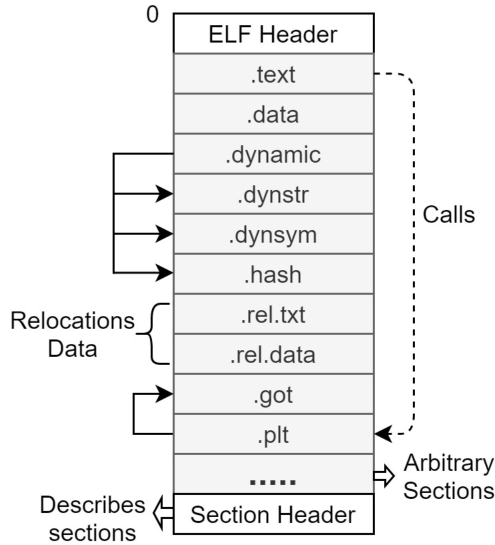


Figure 2.2: ELF Shared Object Structure

execute a program need a program header table, however the presence of a program header table is not a necessity in the case of relocatable files. The individual program headers describe each segment [16]. These segments break down the structure of an ELF binary into suitable chunks to prepare the executable to be loaded into memory. Common [63] types of segments are:

- PT_NULL: Refers to the unassigned segment. `PT_NULL` should always be the first entry of the Program Header Table (PHT).
- PT_LOAD: Refers to the loadable segment. All contents of this segment are loaded into the virtual address space.
- PT_INTERP: Represents the *.interp* section.
- PT_DYNAMIC: Represents the *.dynamic* section.

Figure 2.2 illustrates the structure of the shared object.

2.1.2 Dynamic Linking and Loading

Sections like *.dynamic, .dynsym, .dynstr, .hash, .got* and *.plt* are used to promote the dynamic loading and linking process. The *.dynamic* section is mandatory for creating a dynamic shared object. Apart from that, a dynamic segment entry is also required in the program header table.

The ELF shared objects and relocatable files can be statically as well as dynamically linked to an executable file [74, 29, 64]. In static linking, the linking happens at compile time and the linked ELF object is included in the final executable file, hence it necessitates recompiling the executable if any of its linked objects is changed. On the other hand, in dynamic linking, the linking happens at runtime or load time. The final executable contains the paths of all the ELF objects that are needed to be linked/loaded at runtime. A security mechanism, called address space layout randomization (ASLR) [55], is used to randomize each ELF object's address at load time. The motivation behind the usage of ASLR is to prevent attackers from exploiting known vulnerabilities associated with code loading. For example, to avoid malicious modification at runtime, the executable code is loaded in the read-only address space. Due to ASLR, ELF objects, as well as all other objects that are linked to it, are loaded at random addresses at runtime. While loading the ELF objects in memory, if the code contains a relative address, then relocation of those addresses is needed. In case two processes are trying to share the same code then they cannot do it because the referenced relative addresses are different for both processes, thereby restricting the code sharing in ELF files.

To enable code sharing, the deployment of the position-independent code (PIC) feature is mandatory. The GCC compiler provides a $-fPIC$ flag in order to compile a program into a PIC ELF object/library. ELF objects that are compiled to a PIC binary do not store any absolute addresses in the code section. Instead, all absolute

addresses in the code section are stored in a separate section called *.got* (Global offset table). GOT stores the final (absolute) address of a function call symbol, used in dynamically linked code [12]. The code section uses the relative offsets in the ELF file in order to access the absolute addresses from the GOT. Additionally, the GOT is included in the data section of the ELF file and is not shared between multiple processes. As a result, the loaded ELF object file can share its code section between multiple processes. The function call symbols are called during the program execution of the ELF object. If an ELF file is loaded into memory with a lazy binding option, then the GOT will be writable during the execution because it is located in the data segment (write access) of the ELF file. For a dynamic ELF object to be loaded at any address, it needs to resolve the data and function calls in it. The procedure linkage table (PLT) is used along with GOT to aid this symbol resolution process. This process is explained using an example described below [71]:

```
int foo(void);

int function(void) {
    return foo();
}
```

In case of lazy symbol resolution (RTLD_LAZY), whenever the function `foo()` is called, the process of relocation starts. The code does not call the `foo()` function directly, instead, it calls it with the help of a PLT stub. The call instruction will look like this:

```
    ...
    call 4d0 <foo@plt>
    ...
```

The dynamic loader examines the corresponding relocation entry for the `foo()`

rela.dyn section. The function call `foo()` will have an entry in the *rela.dyn* section as follows:

```
Offset Info  Type Sym. Value   Sym. Name + Addend
0x0200828 0x400000006 R_X86_64_GLOB_DAT 0x0 foo@plt
```

The dynamic loader then sees a R_X86_64_GLOB_DAT relocation type, which instructs the dynamic loader to find the value of the symbol `foo@plt` and put it into address 0x200828. Therefore, the call to the code at 0x4d0 is observed. This 0x4d0 is a PLT stub [74]. The PLT chunk will look as follows:

```
 <foo@plt>:
    jmpq   *0x200382(\%rip)  # 200858 <_GOT_+0x18>
    pushq  $0x0
    jmpq   4c0 <_init+0x18>
```

The first instruction in the PLT chunk jumps (`jmpq`) to the GOT entry of the `foo()` function. As this is the first run, the GOT entry of `foo()` does not contain the real loaded address (absolute) of the `foo()` function. Instead it stores the address of the next instruction in the PLT, i.e., `pushq $0x0`, in this example. This `pushq` instruction will push an offset, which is the index of the `foo()` function's string name in the dynamic string table. Next, it will execute the jump instruction that points to the `_dl_runtime_resolve()` [73] function, which will find the real loaded address of the foo function and update it in the GOT's `foo` entry. Later, when there is a call to `foo()`, the first instruction of the PLT will point to the GOT entry, which already has the absolute address of `foo()`. Therefore, there is no need for indirection to the PLT that will call `_dl_runtime_resolve()` again.

2.1.3 Attributes of the ELF Specification

The ELF specification encompasses several attributes that distinguish it from other existing object file formats like COFF, a.out, etc. Crucial traits of the ELF object format are described in the following sections.

2.1.3.1 Broad Platform Support

The ELF standard provides a set of binary interface definitions that support several operating systems. These interfaces reduce the need for recording and recompiling to smoothen the software development process [16]. Starting with GCC compiler version 2.7, the ELF binary was chosen as the default object file format for Linux [8]. Compilers that run on Unix platforms support ELF as the standard file format [9]. Furthermore, the Tool Interface Standards committee (TIS) chose the ELF standard as a portable object file format. The latest version of Microsoft Windows, i.e., Windows 10, includes a Windows subsystem for Linux that enables ELF support [53].

2.1.3.2 Extensible and Flexible

ELF object files are extensible and are not bound to a specific processor or instruction set [46]. Additionally, they do not rely on a specific word length or data alignment, like big endian or little endian, and support both [52]. Except for the ELF header, every other section and segment has no predetermined position or size and is extremely adaptable. The ELF design provides sufficient flexibility to modify the existing data and add arbitrary optional sections.

2.1.3.3 Code Sharing and Reuse

Unlike the relocatable and executable ELF objects, ELF shared libraries are capable of sharing code between multiple virtual machine instances (Ex. JVM instances) at runtime. This enhances code sharing and decreases the redundancy of code in

memory. For instance, identical methods in multiple programs will have only one copy stored in the shared object. This copy can be consumed by multiple processes at once. This saves memory and does not require loading the same method twice. Therefore, ELF shared objects are best suited for code reuse and sharing, providing size benefits.

2.1.3.4 Position Independent Code (PIC)

As ELF shared libraries use position-independent code (PIC), the code they hold can be loaded at any address in memory [46, 72]. To achieve PIC, the GOT and the PLT are introduced to ELF. Each ELF executable and the shared object associated with it contains a PLT. A PLT is a jump table, which contains entries that indicate an indirect jump to a global offset table entry. PLT permits lazy evaluation, that is, it does not resolve a procedure's address until it is called for the first time.

2.1.3.5 Dynamic Behavior

ELF promotes dynamic linking and loading in shared libraries. The three responsibilities of dynamic linking and loading are to perform relocations, resolve symbols and load the code into main memory. The dynamic linker resolves and relocates all the pointers of the global offset table [46]. To perform automated linking and loading, Linux provides two programs, *ld.so* and *ld-linux.so* [49]. They find and load the shared libraries required by the program to execute. The dynamic linker is also known as the program interpreter and can be accessed from the *.interp* section [49]. Dynamically linked shared libraries allow programs to load and unload routines at runtime, which cannot be achieved using the static ones. To gain access to a dynamic shared library, the *dlopen*(3) function is used. Along with *dlopen*(), *dlsym*(), *dlclose*() and *dlerror*() that implement the interface to the dynamic linking loader [48]. The *dlopen*() function returns a handle that can be

employed with other functions in the *dlopen* API, such as *dlsym*() and *dlclose*() [48]. One of the arguments to *dlopen*() is the mode, which controls the visibility of symbols and relocations. For instance, *RTLD_LAZY* mode performs lazy binding and resolves symbols only when the code referencing it is executed. An example of calls to *dlopen*(), *dlsym*() and *dlclose*() is described below.

```
void *handle;
uint8_t* laddr;
handle = dlopen("/lib/mylib.so.6", RTLD_LAZY);
...
laddr = dlsym(handle, "findlen");
...
dlclose(handle);
...
```

In this case, the dlsym() returns the address where the symbol `findlen` is loaded in the memory. If the symbol is not found in the library then it returns NULL.

2.1.3.6 Tooling Support

The ELF specification is supported by numerous tools, libraries and utilities for manipulating objects and libraries. The GNU binary utilities (*Binutils*) [15] introduce the *readelf* and *objdump* utilities, which can access sections, data and display the information about object files in a human readable format. *Binutils* provides the *objcopy* utility, which can rename sections, update sections, add symbols, copy contents of one object file to another and perform various other operations. The *libelf* library, from the *elfutils* package [14], is capable of reading, modifying and creating the ELF object files. For managing portable dynamic shared libraries, a tool called *GNU Libtool* can be employed. Beyond these tools,

there are several other open source programs and utilities, which enable the manipulation of the ELF object files.

2.2 Eclipse OMR

Eclipse OMR is a set of highly integrated components for runtime developers to build language runtimes [31]. Eclipse OMR is an open-source project and is written in C and C++. All components are language-agnostic, i.e., they are not specific to any language runtime and can support any language runtime depending upon its needs. These components are designed to support multiple instruction set architectures (ISAs) like x86, AMD64, Power, AArch64, etc. Currently, Eclipse OMR components are consumed by runtimes of languages like Java, Ruby, Python, Lua, WebAssembly and others. Eclipse OpenJ9 [28], a prominent JVM, is an example of a language runtime that consumes Eclipse OMR components. The Eclipse OMR compiler, also known as the Testarossa (TR) compiler, is a high performance core component that can be used for just-in-time (TR-JIT) compilation. The JitBuilder API is a library that facilitates runtime developers devising JIT compilers by leveraging the Eclipse OMR TR-JIT component. Other examples of Eclipse OMR components are the port library, the thread library and the garbage collector.

2.2.1 AOT Component

The ahead-of-time (AOT) compilation component of Eclipse OMR is in an initial stage of development [45, 44, 34]. The AOT component utilizes the TR-JIT compiler functionality for AOT method compilation. The Eclipse OMR AOT component consists of core AOT classes:

- `AOTAdapter`: This class is an adapter between multiple components required for performing code loading, storing and updating with current runtime

values. The `AOTAdapter` class has a map (`methodNameToHeaderMap`) that holds a method name as its key and a pointer to the `AOTMethodHeader` object as its value. This map is populated when a method is compiled at runtime. As `methodNameToHeaderMap` holds all method names and its `AOTMethodHeader` pointers, it is used for locating methods at runtime, as well as for serialization.

- `AOTMethodHeader`: The `AOTMethodHeader` class consists of the address of compiled code and the relocation data corresponding to the compiled symbol. `AOTMethodHeader` is generated for every compiled method. It also contains the compiled code size and relocation data size. Additionally, it has functions to serialize and de-serialize the contents of the header to a buffer. `AOTMethodHeader` is supposed to be created and populated after a method is successfully compiled at runtime.

- `AOTStorageInterface`: This class is an interface between the AOT component and the storage container. The interface can be implemented by the runtime developers as per their requirements. Its two main methods are `loadEntry` and `storeEntry`, which are meant to load and store the AOT code and data to a storage container, for example, a shared cache or an object file. During the compile run, i.e., when the program is compiled for the first time, `AOTAdapter` serializes each method using `AOTMethodHeader` and calls the `storeEntry` method to write it to the shared cache. While in the load run, i.e., when the compiled code is already available in the shared cache, `AOTAdapter` loads the method code from the shared cache by calling the `loadEntry` method of `AOTStorageInterface` and de-serializes it using `AOTMethodHeader`.

2.2.2 Relocations Infrastructure

Eclipse OMR introduces the term *external relocations*, which are used to encode runtime dependent data to patch compiled code at load time [45, 44, 34, 27, 26]. These relocations are used for the purpose of ahead-of-time compilation. There are several kinds of external relocations that exist in the OMR AOT component; this research uses a subset of those. Every function that is compiled and called by another function needs its address to be relocated. For example, if `callee_function` is compiled and has been called in another function, then the call instruction will resemble the following:

```
...
call <addr of callee_function>
...
```

Here `addr of callee_function` needs to be relocated whenever the compiled code is loaded into memory. This handling and resolving of external relocations is performed by the AOT relocations infrastructure, which is also used as a part of the Eclipse OMR AOT component. During every method compilation, all necessary external relocations are generated and stored in a binary buffer. This binary buffer is then stored in a storage container and can be loaded back into memory during later runs. Hence, at runtime, when this buffer is loaded, the AOT relocations infrastructure resolves and patches/updates all such addresses. The `AOTAdapter` class provides functions to access the AOT relocations infrastructure. At the time of writing, the AOT relocations infrastructure is a work in progress [34].

2.2.3 Shared Cache

The Eclipse OMR Shared Cache is a revised generic version of the Eclipse OpenJ9 shared classes cache (SCC) [45, 69, 11, 62]. The shared cache was designed and

developed by Thom et al. to serve as a container for storing the compiled code and data. Compiled code and relocations data in Eclipse OMR is generated by the Testarossa JIT (TR-JIT) compiler.

The Eclipse OMR shared cache design allows runtime developers to customize the layout and administration of the cache at a low level. The shared cache introduces the dual notions of initialization and serialization. Serialization is the process of creating a cache region and writing data in it, while initialization is the process of writing data to the cache, only when it is restored from a file or attached to the operating systems memory. The top layer of the Eclipse OMR shared cache structure is `OSCache` [45, 69]. `OSCache` has two derived types, namely, `OSSharedMemoryCache` and `OSMemoryMappedCache`. `OSSharedMemoryCache` is a non-persistent cache, whereas `OSMemoryMappedCache` is a persistent cache. `OSMemoryMappedCache` resides in a file, which can be loaded by other runtime instances using memory mapping capabilities of an operating system, while `OSSharedMemoryCache` uses System V-style shared memory routines. Runtime developers can choose either of these caches as required.

Similar to the Eclipse OpenJ9 shared classes cache, the Eclipse OMR shared cache is allocated as a single contiguous block of memory [45, 11, 69]. The first part in this block is a cache header. The cache header contains information like cache length, header length and cache type. This implies that the shared cache is a fixed-length storage container. The implementation of the shared cache is not fully concrete. Language runtime developers are required to extend and implement several abstract APIs of the shared cache as per the runtime requirement. At the time of writing this thesis, the Eclipse OMR shared cache is also a work in progress.

2.2.4 Extensible Classes

Software variability is the ability of a software component to be effectively extended, modified, or configured to be operable in a definite context [50, 51]. Eclipse OMR leverages software variability to support multiple programming languages on multiple architectures. Dynamic polymorphism allows dynamic typing, but causes performance degradation by adding runtime overhead [25], therefore Eclipse OMR resorts to the notion of static polymorphism. Static polymorphism resolves all inheritance chains and function calls at compile time [50, 51]. Initially, OMR developers planned to use the Curious Recurring Template Pattern (CRTP) [51] to implement static polymorphism, but, CRTP was avoided as it may produce a large amount of template code.

The concept of *extensible classes* in Eclipse OMR demonstrates the use of static polymorphism to achieve software variability. Extensible classes are C++ classes arranged to allow the compiler to find the most concrete implementation of its members at compile time. Extensible classes are tagged with a keyword `OMR_EXTENSIBLE` whenever declared. Most of the reusable code is placed under the top-most class (base class) and more specific code is in the derived class of an extensible hierarchy. In extensible classes, the base class has the OMR namespace, while its derived classes have nested namespaces depending on the project or architecture. For instance, a base class in `codegen`, `CodeGenerator` has the OMR namespace (`OMR::CodeGenerator`), while its derived class in the ARM architecture will have a nested namespace like `OMR::ARM::CodeGenerator`.

To achieve static polymorphism, a single linear hierarchy of extensible classes must be present at compile time. Eclipse OMR enables creating a `Connector` class for each existing extensible classes hierarchy. This connector acts as a liaison between the extensible classes hierarchy in the OMR namespace and external classes in other namespaces that may be extended from this hierarchy [50, 51]. Each time a

class is extended, its connector is declared using the `typedef` and surrounded by `#ifdef` guards. As all classes in the hierarchy will have the same `typedef` connector statement, the guards will ensure that only one of them is defined at a time. The sequence in which these classes are compiled is controlled by `include` paths. Such co-operative employment of connectors, namespaces, guards, `typedef` statements and `include` paths ensure finding the most derived class.

2.2.5 ELF Generation Module

Eclipse OMR supports generation of executable and relocatable (.o) ELF objects [32]. The base class in the ELF generation module is termed `ELFGenerator` and its two derived classes are named `ELFExecutableGenerator` and `ELFRelocatableGenerator`. All these classes are declared and defined in a file named `ELFGenerator` of the `compiler/codegen` directory. Both the derived classes have an `ELFHeader`, and `.text, .data, .symtab, .dynstr` and `.shstrtab` sections. The relocatable ELF object file has an extra `.rela` section to store the relocations data, and the executable ELF object also has the program header. The relocations in `ELFRelocatableGenerator` are "static relocations", which comply with the ELF standard specification. These relocations are different from "external relocations", which are resolved at runtime. In Eclipse OMR, the relocatable and executable objects are "static" in nature. Hence, these files cannot be used in their current form for dynamic loading and linking using the `dlopen` API.

In Eclipse OMR, all generated code and relocations data is stored in the code cache. `ELFExecutableGenerator` and `ELFRelocatableGenerator` are triggered when the code cache destruction is initiated. This CodeCacheManager class contains data structures like `CodeCacheSymbol` and `CodeCacheRelocationInfo`, which store the symbol and relocation information that is needed for ELF object generation [32]. These data structures are populated at the time a method is

compiled. Another issue with `ELFExecutableGenerator` and `ELFRelocatableGenerator` is that it writes the entire code cache into the ELF file. This emits all the contents in the code cache and creates a large ELF object binary [33]. Moreover, the code emitted using `ELFExecutableGenerator` and `ELFRelocatableGenerator` objects, cannot be shared between multiple processes of a runtime. Therefore, the existing ELF generation module is suboptimal for storing, loading and sharing the AOT generated code.

2.3 WebAssembly

WebAssembly (wasm), is a portable code format that allows other languages, like C, C++ and Rust, to run in a web browser at near-native speed [36]. The WebAssembly bytecode format was designed and developed collaboratively by engineers of four major browser vendors: Google, Microsoft, Apple and Mozilla [36]. WebAssembly serves as a compilation target for other languages and several compilers exist to enable it. For instance, the Emscripten [17] compiler toolchain for C/C++, or the Rust compiler, allows transforming the source code written in C/C++ and Rust into WebAssembly bytecode format (*.wasm*). Emscripten and the Rust compiler rely on the low-level virtual machine (LLVM), a set of compiler and toolchain technologies. Using the WebAssembly JavaScript APIs, WebAssembly modules can be loaded into JavaScript and functionality between the two can be shared. The usage of WebAssembly along with JavaScript makes it possible to reap the benefits of both worlds. WebAssembly is relatively new, still evolving and has attracted significant interest from the web community.

2.3.1 WABT

The WebAssembly Binary Toolkit (WABT) is a collection of tools that are used to manipulate WebAssembly files [70]. For instance, WABT provides tools to convert WebAssembly text format into WebAssembly binary format and vice versa. WABT also includes a WebAssembly interpreter (*wasm-interp*) written in C++. This stack-based interpreter decodes and executes a WebAssembly binary file (*.wasm* extension).

2.3.2 wasmjit-omr

The wasmjit-omr compiler is a WebAssembly Just-in-Time (JIT) compiler and is derived using Eclipse OMR [7]. The wasmjit-omr compiler is written using the Eclipse OMR JitBuilder library.

2.3.3 wabtaot

Wabtaot [45, 30, 42] is a WebAssembly Ahead-of-time (AOT) compiler that consumes the JitBuilder framework and the OMR AOT component of Eclipse OMR. To compile a wasm module, wabtaot uses the Testarossa JIT compiler, which is accessed through the JitBuilder API. *Wabtaot* uses the AOT relocations infrastructure to handle the relocations. This AOT compiler takes a wasm binary file (*.wasm*) as input, compiles it and generates native code. The wabtaot compiled code and data is stored in a wabtaot-specialized implementation of the Eclipse OMR shared cache. The wabtaot compiler performs operations like compiling, storing, loading and relocating, by using the following 4 methods from the Eclipse OMR JitBuilder API for its compilation process:

- `loadCodeEntry`: This method calls the `loadEntry()` method of the `AOTStorageInterface` class, which tries to load a method, if it already exists. After loading the serialized buffer, it de-serializes the buffer and

registers the method in the AOT relocations infrastructure.

- `storeCodeEntry`: This method successively calls the `storeEntry()` method of the `AOTStorageInterface` class, which stores the serialized AOT code and data buffer into the shared cache, after it is compiled.

- `internal_compileMethodBuilder`: It compiles the method using the Eclipse OMR Testarossa JIT backend compiler and generates the compiled code and relocations data.

- `relocateCodeEntry`: This method resolves all relocations at runtime using the Eclipse OMR's AOT relocation infrastructure.

Both wasmjit-omr and wabtaot are used within the WABT runtime.

Chapter 3

Review of Runtimes

The idea of employing AOT compilation to boost the start-up time of a virtual machine is an open-ended research topic. With this, the discussion of persisting and loading back the native compiled code from a storage container is also crucial. On a few occasions, members of the WebAssembly community have advocated for the usage of an ELF object as a container format [35]. Their discussion mainly focuses on the characteristics of an ELF object, its advantages and the downsides of using it in a WebAssembly language runtime. Likewise, a similar discussion took place about the possibility of using an ELF object for AOT compilation in Eclipse OMR [39]. Such ongoing exchange of views indicates continued interest in ELF object files. AOT compilation is adopted by several well-known language runtimes such as the Java Virtual Machine (JVM), Mono, the Android Runtime (ART), V8 and certain Python implementations, like Cython [61, 37, 18, 58]. In this section, a handful of state-of-the-art runtimes are reviewed, including few WebAssembly runtimes, and their AOT compilation techniques. The primary emphasis will be on the storage of AOT compiled code and data. Table 3 lists runtimes with their backend AOT compilers and storage containers.

Runtime	Language	Backend Compiler	Storage
Mono	.NET	LLVM	ELF
HotSpot	Java	Graal JIT	ELF
ART	Dex Bytecode	dex2oat	ELF-like
OpenJ9	Java	Testarossa	Shared Classes Cache
Zing	Java	Falcon (LLVM-based)	Falcon Cache
V8	JavaScript & WebAssembly	Liftoff and TurboFan	Code Cache & Resource cache
Wasmer	WebAssembly	Single-pass or Cranelift or LLVM	ELF
Wasmtime	WebAssembly	Cranelift	cwasm
Lucet	WebAssembly	Cranelift	ELF

Table 3.1: Runtime Systems and their AOT features.

3.1 Mono Runtime

Mono Runtime is an open-source implementation of Microsoft's .NET framework [61]. Mono AOT Compilation is a feature of the Mono Runtime Code Generator. The AOT compiler in Mono uses the object format native to the target platform to store the AOT-compiled code [60]. On ELF supported platforms, it generates a shared object file (.so), also called an AOT image. However, on other platforms it generates an assembly (.s) file, which in turn can be assembled and linked into a shared object file.

For method compilation, Mono uses the JIT compiler, which generates compiled code and relocation patches [60]. The native code generated needs to reference various functions and structures, whose address can be known only at runtime. JITed code can directly patch the address into the native code, but AOTed code needs to

go through an indirection, which is deemed as a performance overhead [60]. This indirection is done through a table called the Global Offset Table (GOT). The GOT and the Procedure Linkage Table (PLT) used in the Mono Runtime are similar to the ELF Specification [60]. The AOT compiler considers all function calls in the program as one type of patch and stores them into the PLT. Everything, other than function calls, is regarded as "other" types of patches, which are stored in the GOT. Other than storing the position independent code generated by the AOT compiler, the AOT image also stores cached metadata. The cached metadata involves a variety of information, like instance size, type initializer, etc., that is useful for class loading at runtime. Computing the cached metadata is time consuming and requires creation of runtime data structures. Hence, the required information is computed during AOT compilation and stored in the AOT image, into a `class_info` array. In Mono AOT, a shared object containing only the metadata can be produced using the `metadata-only` compilation option. The version of the AOT file format and the list of assemblies referenced by the AOT module are stored in the AOT image. Each AOT image contains a global symbol `mono_aot_file_info` that points to the `MonoAotFileInfo` structure. The `MonoAotFileInfo` structure holds pointers to all the AOT data structures. Mono AOT handles two types of methods, normal and extra. Normal methods are methods present in the `METHODDEF` table, whereas extra methods are either runtime generated methods or inflated generated methods [60]. A normal method is identified by a method index stored in the method metadata table. However, extra methods are identified using arbitrary numbers.

3.2 Oracle HotSpot

Another prominent Java runtime is the HotSpot JVM by Oracle [37]. Static AOT compilation in the HotSpot JVM employs the Graal framework for generating the

AOT code [23]. The code container utilized for storing AOT code in the HotSpot JVM is the ELF shared library (.so) file. These ELF shared libraries are produced using the *libelf* library from the *GNU elfutils* project [43]. Hotspot uses the *javac* tool to compile a source file and generate a *.class* file. For AOT compilation, the tool *jaotc* is used to generate the native code of the processed class file. Commonly, the code generated in the HotSpot AOT compilation is treated as an extension of the existing HotSpot Code Cache. To AOT compile only specific methods, *jaotc* provides a *compileOnly* flag, whereas to exclude methods from a compilation it offers an *exclude* flag.

Tiered Compilation (TC) mode in HotSpot is able to collect and store profiling information along with the code. The HotSpot JVM introduces an interesting notion of class fingerprinting [43]. This technique stores the fingerprint of each class after AOT compilation in the *.data* section of the ELF shared libary. The objective of this technique is to determine if the class has been changed after AOT compilation. This is done by matching the current fingerprint of the class with the one stored in the shared library during loading. The relocation data is stored similarly to Mono in its reliance on the GOT and PLT tables [68]. The HotSpot JVM needs the same runtime configuration for compilation and execution. In JDK 10 SE, AOT compilation is an experimental feature and is supported only on Linux-x64 [23].

3.3 Android Runtime

Android Runtime, also known as ART, is an application runtime environment used by Android OS [57, 58]. ART was introduced in Android 5.0 Lollipop, thereby replacing the Dalvik Virtual Machine (DVM). In Android, programs written in Java are first compiled to Java bytecode (.class) and then to DEX bytecode (.dex) format. DEX is a Dalvik executable file format that has been borrowed from the DWARF3

Specification [57, 58]. ART compiles DEX bytecode and generates its native code. A key difference between ART and Dalvik VMs is in their compilation strategies. ART is equipped with an AOT compiler, whereas Dalvik VM uses JIT compilation. During the applications installation phase, ART uses an on-device *dex2oat* tool to statically convert DEX bytecode into native code. The outcome of the *dex2oat* operation is an Of-Ahead-Time (.oat) file. These .oat files are actually embedded in ELF object files. As AOT compilation happens during the installation phases, there is no need for JIT compilation during application startup, which reduces the startup time and CPU utilization.

3.4 Eclipse OpenJ9

Eclipse OpenJ9 is an open-source JVM implementation based on the IBM J9 JVM. It supports JIT compilation as well as AOT compilation and stores the compiled code in a common container format termed the Shared Class Cache (SCC) [18]. In addition to compiled code, the SCC also stores metadata required for execution and linking. Typically, the SCC is located in shared memory and exists beyond the lifetime of the JVM. Also, it is a fixed-size cache that allows storing AOT code and data.

In Eclipse OpenJ9, the AOT compiler is automatically activated upon enabling the SCC [19, 10]. The JVM splits and stores a class in immutable (read-only) and mutable (writable) portions [18]. Caching AOT method data reduces the need for JIT compilation, as all subsequent runs can utilize the cached data rather than JIT compiling it again. The JVM decides the extent of AOT methods to be stored in the shared cache. Typically, the AOT method data is around 10% of the total class shared data. There is no limitation on the number of SCCs and no particular JVM owns any SCC. However, a JVM can connect and read from only one SCC at a

time. The SCC locates a fixed-sized AOT relocation header in addition to the AOT code. The relocation records, along with validation records, are packed contiguously into fixed-size blocks in the AOT relocation header [68]. Another benefit is that the persistent SCCs can be moved between machines having the same operating systems and hardware specification. The SCC improves the startup time of the VM in its subsequent runs as the native code stored in the SCC is loaded [10, 13].

3.5 Zing Runtime

The Zing virtual machine is a HotSpot derived JVM, developed by Azul Systems [65]. The Zing virtual machine is complemented by the Azul ReadyNow! technology to extend the power and capabilities of the Zing runtime [67]. It introduces a compile stashing technique to resolve the warm up issues and improve the start time of the VM. Compile stashing is the Zing equivalent of AOT compilation. For AOT compilation, Azul uses a LLVM-based JIT compiler named Falcon [66]. The bytecodes are compiled and stored in the Falcon cache by the Falcon JIT compiler [67]. By default, the size of the Falcon cache is 10 GB, however the size can be adjusted using command line options. This provides flexibility to the user in adjusting the cache size based upon system memory. Falcon cache does not evict its contents automatically and continues accumulating until it reaches its maximum size.

3.6 V8

V8 is an open-source high-performance JavaScript and WebAssembly runtime, developed by Google [41]. V8 is written in C++ and runs on Windows 7+, MacOS 10.12+ and on Linux systems, which use x64, IA-32, ARM, or MIPS processors. For WebAssembly compilation, V8 deploys two compilers: Liftoff and TurboFan.

Liftoff is an one-pass compiler, which iterates over WebAssembly code and emits machine code for every instruction independently [41]. One-pass compilers are known for fast code generation, but limit themselves in performance optimizations. Liftoff compiles WebAssembly modules to execute them as early as possible. Once Liftoff compilation is finished, the compiled WebAssembly module is returned to JavaScript.

After Liftoff compilation, V8 re-compiles all functions from the same WebAssembly module using the TurboFan optimizing compiler, which runs in the background to produce optimized native code for both JavaScript and WebAssembly [41]. As TurboFan is a multi-pass compiler, it generates multiple internal representations of the compiled code before emitting it. By leveraging these multiple representations, TurboFan offers more optimizing opportunities and better register allocations. TurboFan compiles WebAssembly modules function by function. Once a function is compiled, it replaces the previously generated Liftoff code with newly optimized high-quality TurboFan code. TurboFan consumes considerable time to compile and generate native code, which introduces a performance penalty. Therefore, to avoid this penalty, AOT compilation is introduced in V8. When TurboFan compiles large WebAssembly modules, it stores the compiled code in a cache [40]. For instance, Chrome saves the TurboFan generated code in its code cache, then loads and de-serializes it in later runs. Loading and de-serializing the compiled code in later runs avoids both Liftoff and TurboFan compilation, thereby improving the startup time and power consumption of V8. This process of storing into the cache and loading back in later runs is AOT compilation in V8. V8 also caches the *.wasm* module in a cache known as the resource cache. This caching is done to save time required to load the module from the network each time. As caches consume space from user's machine, they are limited to few hundred megabytes. Currently, the WebAssembly code cache in Chrome V8 is 150 MB (max limit) for desktop

browsers.

3.7 Wasmer 1.0

Wasmer 1.0 is a server-side WebAssembly runtime released in January 2021 with a stabilized API [5, 4]. Wasmer generates ELF binaries that can run on operating systems like Linux, macOS, Windows and also on web browsers. One of the interesting features of the Wasmer runtime is its support for multiple backend compilers. Three backend compilers that can be plugged into Wasmer are Single-pass, Cranelift (default) and LLVM [4]. Other than multiple compiler backends, it also has dual engines: a JIT engine, which compiles and stores the WebAssembly modules in memory; and a native engine, which compiles the WebAssembly modules ahead-of-time and generates shared objects. The native engine is its AOT compiler component. In later runs, the shared object generated by the native engine is loaded into memory and then executed. A WebAssembly module in Wasmer is compiled as follows:

```
$ wasmer compile --native sample.wasm -o sample.so
```

After compiling, the sample.so file is loaded and executed as follows:

```
$ wasmer run sample.so
```

3.8 Wasmtime

Wasmtime is a standalone WebAssembly runtime developed by the Bytecode Alliance project [59]. Wasmtime runs WebAssembly modules outside of the web. The backend compiler in Wasmtime is the Cranelift JIT that generates high-quality machine code at runtime. Wasmtime also supports AOT compilation. It stores the AOT compiled

WebAssembly module in a `.cwasm` extension file. This `cwasm` binary is a Wasmtime-specific AOT code stage container. On inspecting the contents of the `cwasm` binary structure, it has a resemblance with ELF file structure, but it is not an ELF file. The commands to AOT compile and execute a WebAssembly module are as follows:

```
$ wasmtime compile foo.wasm
$ wasmtime foo.cwasm
```

The first command will compile the foo.wasm module and generate a file with the same name, but a `.cwasm` extension. This generated file is then directly executed using Wasmtime. The environment running the AOT-compiled `.cwasm` module should be the same as the host environment on which the module was AOT-compiled.

3.9 Lucet

Lucet is a native WebAssembly compiler and a runtime, developed by Fastly in collaboration with the Bytecode Alliance [38]. Lucet supports the WebAssembly System Interface (WASI) [3], a modular API for WebAssembly. In Lucet, the compiler is responsible for compiling the WebAssembly module into native code. Whereas, the runtime manages all resources and traps runtime faults. The Lucet compiler backend is built on top of the Cranelift JIT code generator [38]. The main idea behind the design of Lucet is its default support for ahead-of-time compilation of WebAssembly modules. The native code generated by the Lucet AOT compiler (Cranelift) is stored in an ELF-based shared object. The commands for compilation and execution are as follows:

```
$ lucetc-wasi hello.wasm -o hello.so
$ lucet-wasi hello.so
```

3.10 Summary

Wasmer, Lucet and Hotspot are runtimes in which the ELF object has the ELF magic number at the beginning of the file. Whereas, Wasmtime, ART and Mono replace the ELF magic number with their own magic number. The shared object implemented in this research contains the ELF magic number, thereby identifying itself as a proper ELF object. The major difference between the ELF shared object developed in this research and other ELF shared objects used in these runtimes is the process of resolving the symbol addresses at runtime. Most of these runtimes use the conventional approach, which includes using the GOT and PLT. The existing AOT relocations infrastructure of Eclipse OMR is utilized to resolve them at runtime. Also, rather than focusing on a single language runtime, the aim is to provide a generic AOT-Specific ELF object generator for Eclipse OMR. The attributes like wide tooling support, dynamic features and portability are the reason to choose an ELF object-based approach over a cache-based approach.

Chapter 4

Design and Implementation of Shared Objects

The existing WebAssembly AOT compiler (*wabtaot*), developed by CAS-Atlantic, is a fairly new component [45, 42]. This wabtaot compiler uses the Eclipse OMR AOT component and the AOT relocations infrastructure to resolve the symbol addresses at runtime. To demonstrate the usage of ELF objects using the Eclipse OMR AOT component, the existing WebAssembly AOT compiler (*wabtaot*) is chosen as the use-case. The existing design of *wabtaot* is modified in order to address the need to generate an ELF shared object. In this solution, an ELF-based object generator is implemented in Eclipse OMR to store and load the AOT generated code and data. This storage option is an alternative to the existing AOT storage support, the shared cache of Eclipse OMR. The design of AOT-specific shared object generation using the *wabtaot* compiler with Eclipse OMR is showcased in Figure 4.1. The goal is to allow language runtimes that generated compiled code and relocation data using the Eclipse OMR's AOT component to generate an ELF shared object, which can be loaded in later runs using the dlopen API. This implementation is a proof-of-concept demonstrating the use of an ELF shared object as an AOT code storage container

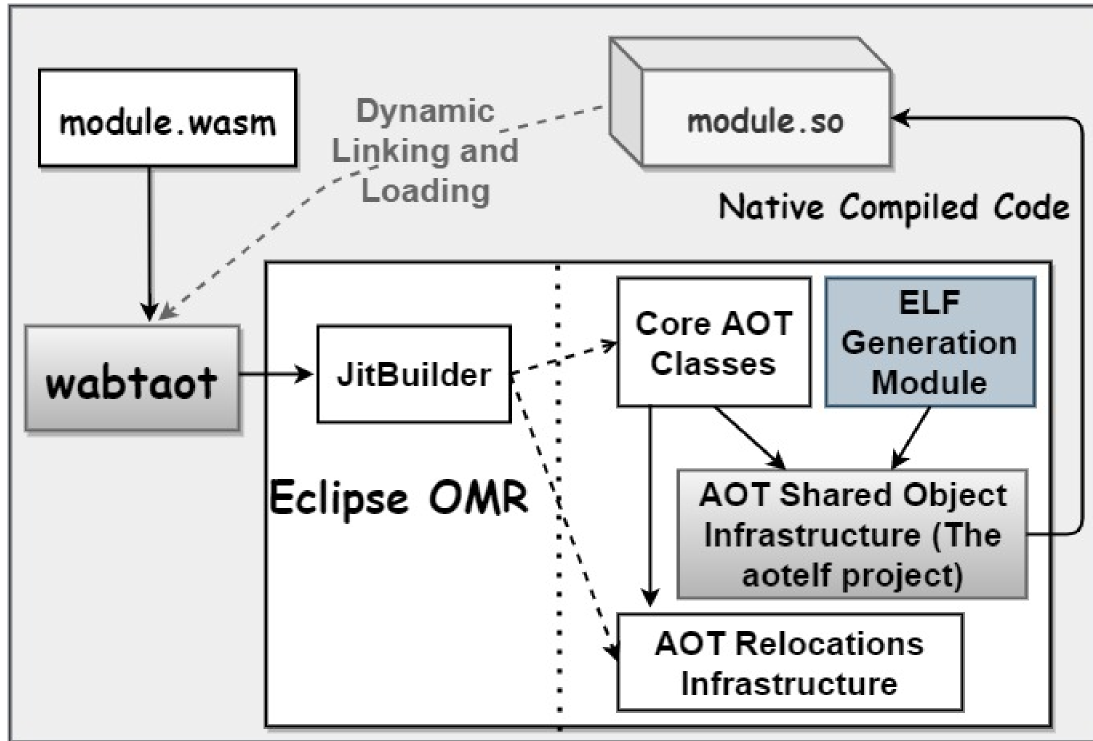


Figure 4.1: Design of AOT-ELF Shared Object Generation within *wabtaot* runtime with Eclipse OMR.

in Eclipse OMR. A detailed explanation of the design and implementation is in the subsections below.

4.1 Variability in ELF Generation Module

The existing ELF generation class hierarchy in Eclipse OMR does not utilize all the instruments for achieving variability. For instance, `ELFGenerator` is sub-classed by `ELFExecutableGenerator` using dynamic polymorphism. As mentioned in Section 2.2.4, dynamic polymorphism introduces runtime overhead, which can be avoided by using static polymorphism. To provide an opportunity for language runtime developers to extend to multiple target architectures, operating systems and also in projects outside the OMR namespace, all current classes of this module are transformed into extensible classes.

In the newly proposed extensible class hierarchy, the `ELFGenerator` class is the base class of the new extensible hierarchy. As seen in Listing 4.1, an `ELFGeneratorConnector`, define guards, typedefs and namespaces are introduced with the `ELFGenerator` class,.

Listing 4.1: Extensible Class Connector Representation

```
#ifndef OMR_ELFGENERATOR_CONNECTOR
#define OMR_ELFGENERATOR_CONNECTOR
namespace OMR { class ELFGenerator; }
namespace OMR { typedef
                OMR::ELFGenerator ELFGeneratorConnector; }
#endif //OMR_ELFGENERATOR_CONNECTOR
```

`ELFExecutableGenerator` and `ELFRelocatableGenerator` classes are also transformed into extensible classes, thereby having their own connectors, typedefs and define guards. `ELFExecutableGenerator` is assigned an `ELFExecutableGeneratorConnector`, whereas `ELFRelocatableGenerator` is assigned an `ELFRelocatableGeneratorConnector`. Under this new design, both classes can spawn their own subclass tree. For example, the listing 4.2 demonstrates how the `ELFExecutableGenerator` class extends `ELFGeneratorConnector`, eschewing dynamic polymorphism.

Listing 4.2: Demonstration of Connector Class Extension

```
class OMR_EXTENSIBLE ELFExecutableGenerator
            : public OMR::ELFGeneratorConnector
```

Additionally, to allow code sharing and dynamic linking and loading, a new extensible class, `ELFSharedObjectGenerator` is derived from the class `ELFGenerator`. `ELFSharedObjectGenerator` is the base class of the extensible class hierarchy that uses `ELFSharedObjectGeneratorConnectors` and is the most

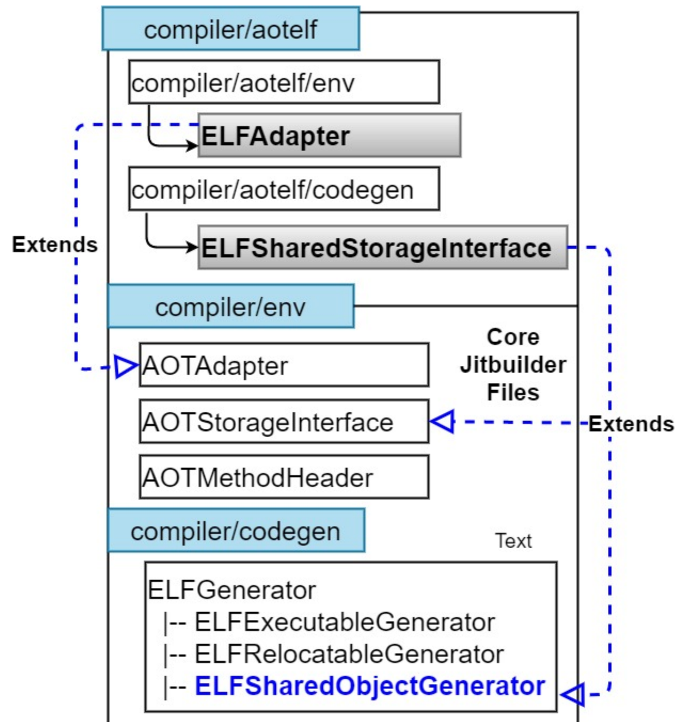


Figure 4.2: The aotelf Compiler Project Directory Structure using Extensible Classes

"generic" version. As seen in Figure 4.2 `ELFSharedObjectGenerator` is a part of the `ELFGenerator` class file in the compiler's *codegen* project.

4.2 The aotelf Project

In the present scenario, the methods in AOT component classes are able to store and load *wabtaot* generated code and data from a variation of the Eclipse OMR shared cache. For instance, the `loadEntry` and `storeEntry` methods of `AOTStorageInterface` class are responsible for loading and storing AOT code and data from the shared cache. The objective of this research is to store and load the AOT code and data in an ELF shared object, instead of the shared cache, by providing an alternative implementation of `AOTStorageInterface`. By default, all AOT component classes in the `compiler/env` directory and the `ELFSharedObjectGenerator` class in the `compiler/codegen` directory are used as

part of the compiler, and consequently, JitBuilder (see Figure 4.2). Modifying these classes would mean interfering with all the projects that use JitBuider. Therefore, to reduce modifications to the JitBuilder code, the *aotelf* project is created, which allows using an ELF shared object as an example implementation of a storage container for the Eclipse OMR AOT component. The *aotelf* project generates an AOT-specific ELF shared object by extending the AOT component classes and the ELF generation module. Additionally, new methods are introduced to enable the shared object generation and loading.

4.2.1 Extended AOT Component

The first component of the *aotelf* project involves extending the AOT component classes [44, 45] from Eclipse OMR and providing an alternative, ELF-specific implementation to its methods in the *aotelf* project. This alternative implementation is enabled by overriding the methods of the AOT component classes to store data into an ELF shared object. To achieve that, `AOTAdapter` and `AOTStorageInterface` are overridden by the new ELF-specific `ELFAdapter` class and the `ELFStorageInterface` class. Some of the AOT component classes, for example, `AOTMethodHeader` did not require any modification and are used as-is.

The second component of the *aotelf* project is the AOT-specific shared object generator. In the shared cache version of wabtaot, after each method is compiled, it is written immediately to the shared cache using the `storeEntry` method of `AOTStorageInterface`. Such an approach could not be adopted for ELF shared object generation, because it would require adjusting the offsets of every section after each method is compiled and added to the ELF object. To avoid this, ELF shared object generation is triggered only when all methods are compiled and their AOT code and data are available. Unlike the shared cache's `storeEntry`, the new `storeEntry` method defers writing to the shared object and stores the AOT data in

the `ELFDataMap` data structure. The elements of the `ELFDataMap` have the compiled method names as their key and their corresponding `AOTMethodHeaders` as their value. While the `AOTAdapter`'s `methodNameToHeader` map contains all the methods compiled or loaded during this instance of runtime, `ELFDataMap` stores the AOT data of the methods that are only intended to be written to the shared object.

4.2.2 Extensible Classes + Multiple Inheritance

Initially, `ELFDataMap` was designed as a part of `ELFStorageInterface`, located in the *aotelf* project. In later stages, it was observed that the data in it is directly used by the `ELFSharedObjectGenerator` class, which is a part of the Eclipse OMR compiler. This means that `ELFSharedObjectGenerator` has to access a member of `ELFStorageInterface`, which indicates a design conundrum that exposes the more general project to the subproject's implementation details. Overall, `ELFSharedObjectGenerator` is supposed to generate an OMR ELF shared object, whereas the `ELFStorageInterface` class should contain an implementation of `AOTStorageInterface`, which is manipulated by the *aotelf* AOT component, allowing for the functionality of loading from and storing to a shared object.

The task of combining the two class functionalities is solved by introducing multiple inheritance using extensible classes, which is also an engineering contribution of this research, since it was not explored before in the literature. `ELFSharedObjectGenerator` in the *aotelf* project derives from the two classes, located in the file `ELFSharedStorageInterface` to denote the composition of two classes.

As a result of multiple inheritance, all members in `ELFSharedStorageInterface` can now access `ELFDataMap` and can also have ELF shared object generation functionality. A demonstration of multiple inheritance in extensible classes is shown in Listing 4.3.

Listing 4.3: Multiple Inheritance using Extensible Classes

```
#ifndef ELF_SHOGENERATOR_CONNECTOR
#define ELF_SHOGENERATOR_CONNECTOR

namespace ELF { class ELFSharedObjectGenerator;  }
namespace ELF { typedef ELFSharedObjectGenerator
                    ELFSharedObjectGeneratorConnector; }
namespace ELF { typedef ELFSharedObjectGenerator
                    AOTStorageInterfaceConnector; }
...
...
class OMR_EXTENSIBLE ELFSharedObjectGenerator
   : public OMR::ELFSharedObjectGeneratorConnector,
     public OMR::AOTStorageInterfaceConnector
```

The directory structure of the *aotelf* project can be seen in Figure 4.2. The folders and the include paths are set to allow locating the right classes at the stage of the project compilation, according to the extensible classes concept. For example, in CMake, using the "BEFORE" keyword allows "shadowing" the Eclipse OMR JitBuilder core classes with extended implementation in the *aotelf* project. For example, `AOTAdapter` located in `aotelf/codegen/ELFAdapter` shadows `AOTAdapter` in `codegen/env`.

4.2.3 Compilation in wabtaot

As seen in Figure 4.1, the *wabtaot* compiler takes a wasm module as an input, and produces an ELF shared object file as an output, while also executing the program. When a wasm module is executed for the first time, *wabtaot* compiles all the methods

and stores the AOT code and data in a shared object. However, when the same module is executed for the second time, *wabtaot* fetches the already existing shared object and loads it. Hence, the first run is called a "compile run" and subsequent runs are called "load runs". The command to compile using *wabaot* is as follows:

```
./wabtabot < module.wasm >
```

As per the design and implementation, the shared object, if available, is loaded into the memory at the beginning. Additionally, if a method is compiled and new code is generated, a shared object needs to be regenerated. Since such loading and generating functionality is not currently present in either core AOT classes or in the JitBuilder library, the JitBuilder API is extended with two new methods, `loadObject` and `generateObject`.

- `loadObject`: The `loadObject` method will access the *aotelf* class members and load the shared object into the virtual address space using the `dlopen()` method. `dlopen()`'s handle is recorded in `ELFSharedStorageInterface` so that all future `dlsym()` calls can use the same.
- `generateObject`: The `generateObject` composes an ELF object structure.

When *wabtaot* tries to load the shared object at the beginning of compilation using `dlopen`, there can be two possible scenarios based on the results returned by the `dlopen()` function:

- `dlopen()` successful: If `dlopen()` is successful, it will return a handle to the loaded shared object. Subsequently, all methods are loaded one-by-one into the Eclipse OMR code cache using the `loadCodeEntry` method of the JitBuilder API. The `loadCodeEntry` method calls the `loadEntry` method, which will use the recorded handle pointer and apply the `dlsym()` method to retrieve the loaded (absolute) address of a method. The `loadEntry` method is implemented

to use the persisted handle pointer and apply the `dlsym()` method to retrieve the loaded (absolute) address of a method.

Listing 4.4: dlopen() and dlsym() Function Calls

```
_handle = dlopen(fileName, RTLD_LAZY);
addr = (uint8_t*) dlsym(_handle, key);
```

As seen in Listing 4.4, the key contains a symbol whose address is fetched by the `dlsym()` function using the `dlopen()` handle pointer. If the key is not present in the loaded object, the dlsym will return a null pointer. This also means that the input module has new methods, which are not present in its previously generated shared object. Hence, the new method is compiled and the wabtaot's compilation status is set to "emit" for the module. The "emit" flag indicates that after compilation ends, an ELF shared object must be generated for that module, thereby achieving incremental linking.

- `dlopen()` unsuccessful: The wabtaot's compiler status is set to "emit" for the module that was attempted to load. Thereafter, every method in the module is compiled and stored in `ELFDataMap` using the `storeEntry` method of `ELFSharedStorageInterface`.

Once either of these scenarios are completed, *wabtaot* resolves all relocations by calling the relocation infrastructure of Eclipse OMR and then performs execution. After the execution ends, if the flag is set to "emit", then *wabtaot* triggers the shared object generation using the newly-introduced `generateObject` method in the JitBuilder API.

4.2.4 AOT Data Serialization

An AOT buffer is formed using the members of the `AOTMethodHeader`. Figure 4.3 shows the content of an AOT buffer. The shared cache approach serializes each AOT

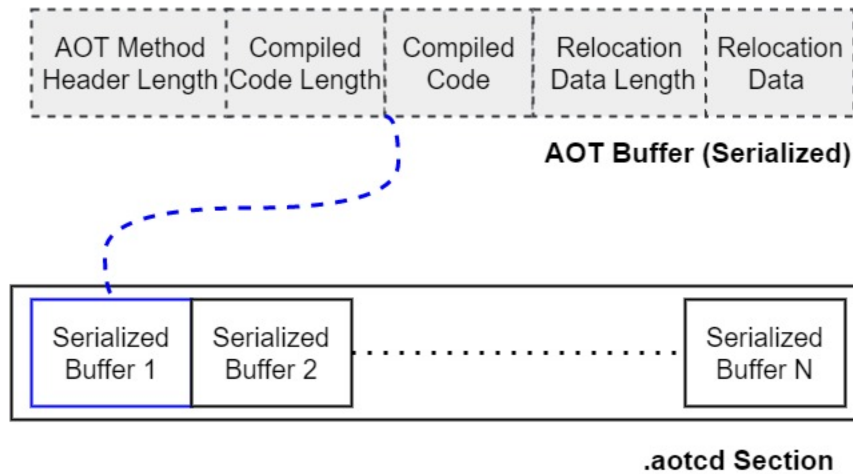


Figure 4.3: Depiction of Serialization into ELF Shared Object using `AOTMethodHeader`

buffer into the shared cache after each method is compiled. Serializing every AOT buffer into the shared object immediately after compilation would require updating section offsets of the shared object after every write operation. Therefore, by using the `AOTMethodHeaders` data, a consolidated buffer is created, which stores the AOT code and data as shown in Figure 4.3. This buffer is written to a shared object in a single `fwrite()` operation.

4.2.5 AOT Shared Object Composition

After the consolidated buffer is created, `generateObject` starts composing the shared object. At first, the ELF shared object headers and section headers are initialized. The ELF header is written, followed by the program header table. The type field in the ELF header is set to `ELF_DYN`, which is the type identifier for dynamic shared objects. Furthermore, the program header table is loaded with a new `PT_DYNAMIC` entry. By doing so, the ELF object is a dynamic shared object that can be loaded at runtime using the dlopen API.

The `ELFDataMap` is iterated to retrieve the contents of each `AOTMethodHeader` and serialize (see Figure 4.3) the compiled code and relocations data (AOT Buffer) of

each method in the shared object. As serialization of the compiled code as well as relocations data is done into the same section, this new arbitrary section is termed as the *.aotcd* section. As a result, the flexibility of the ELF design is leveraged by introducing an arbitrary section called *.aotcd* to store AOT data. This section is given read and execute permissions. The existing approach in `ELFExecutableGenerator` and `ELFRelocatableGenerator` emits the whole Eclipse OMR code cache into the `.text` section, which results in large-sized object files [33]. The main benefit of such an approach is a small-sized object file output with just AOT-specific code and data.

Subsequently, the metadata of all methods is stored in the dynamic symbol table that is in the *.dynsym* section and all method names are stored in the *.dynstr* section. This metadata information is extracted from the `ELFDataMap`. An important addition to the ELF shared object is the *.hash* section. The *.hash* section is mandatory in a dynamic shared object [24]. The hash functionality helps locate the symbols from the dynamic symbol table, thereby accelerating the search time. A System V-style hash table is implemented in compliance with the ELF specification and is written in the *.hash* section. After the *.hash* section, a blank *.data* section is written, followed by a *.dynamic* section. Finally, the *.dynamic* section entries are populated with details of the dynamic symbol table, dynamic string table and hash table. The structure of the ELF file is pictured in Figure 4.4. Eventually, the presence of all these sections enables the `dlopen` API to dynamically load the ELF file.

Thereafter, the generic code components of `ELFSharedStorageInterface` code are determined and moved to `ELFSharedObjectGenerator`. For instance, to create the `.hash` section, a function called `getNoOfBuckets` is implemented. As this function does not have any AOT-specific dependency, it is moved to `ELFSharedObjectGenerator` in the `compiler/codegen`. The composition of sections for the AOT-specific shared object is self-contained, but the composition of

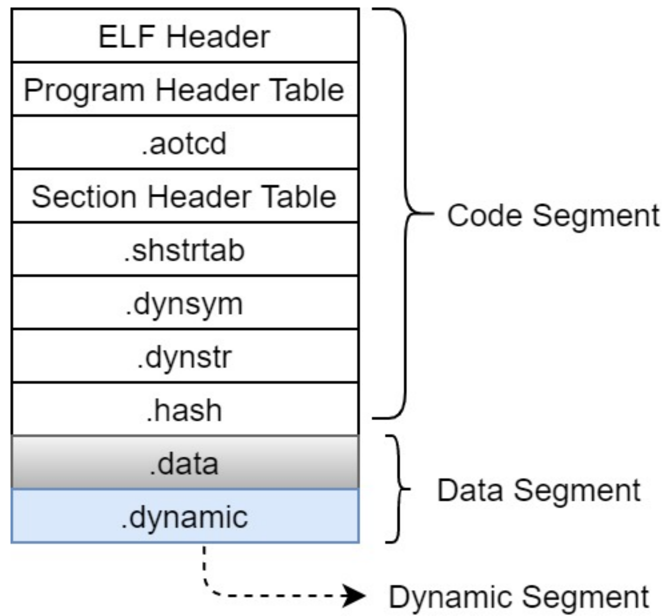


Figure 4.4: Structural Composition of an AOT Shared Object

the sections in the generic `ELFSharedObjectGenerator` is left abstract for the language-runtime developers to extend and compose it, as per their requirements.

4.3 Wabtaot Modes and Shared Library Approach

The shared library implementation uses the same shared object functionality described above, however it is able to compile multiple modules and store them in a single centralized shared object, which I call a shared library. As seen in Figure 4.5, the wabtaot compiler takes multiple modules as input and generates a common shared library called `wasmaot.so`. The implementation of a shared library approach only requires changes in the wabtaot infrastructure and not in Eclipse OMR, as all the existing functionality is reused as-is. With the introduction of such an approach, two modes are incorporated in the wabtaot compiler, called the exclusive mode and the inclusive mode. These modes are applied as flags:

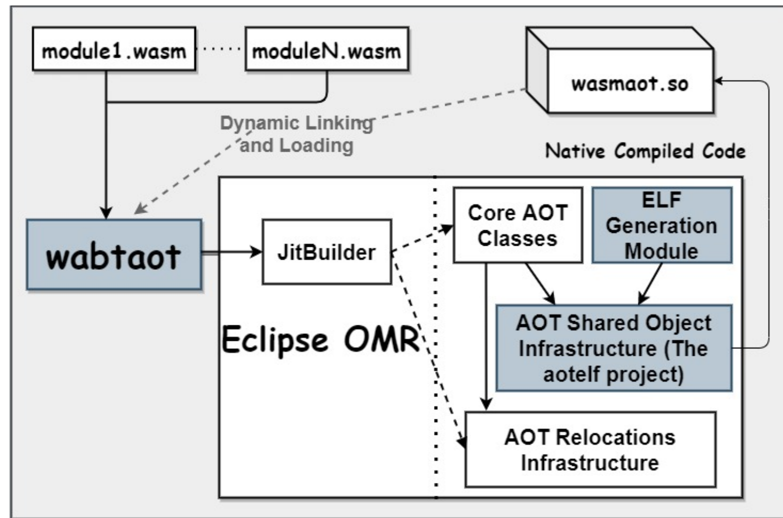


Figure 4.5: wabtaot Shared Library Approach

`--inclusive` or `--exclusive`. The exclusive mode is a shared object-based approach, where an object is generated for each input module. The command for exclusive mode is as follows:

```
./wabtaot --exclusive sample.wasm
```

On the first run, this command will compile the module, generate a `sample.so` shared object and execute it. On the second run, this command will load the `sample.so` object and perform execution. The other mode, i.e., the inclusive mode, is based on the shared library approach. The inclusive mode should be applied when multiple modules are destined to share an object (library). The commands that describe the exclusive mode are as follows:

```
./wabtaot --inclusive sample1.wasm sample2.wasm sample3.wasm
./wabtaot --inclusive sample.wasm
```

The first command will compile the module, generate a `wasmaot.so` shared object and execute it. The second command will load the `wasmaot.so` object and perform execution. The inclusive mode looks for the shared library, whereas the exclusive mode looks for a specific shared object.

4.4 Summary

This chapter elaborated the design and implementation of the ELF shared object generator in Eclipse OMR. It presented the *aotelf* project, which extended the AOT component along with the ELF generation module to produce an AOT-specific ELF shared object. Furthermore, the usage of the *aotelf* project for AOT compilation in *wabtaot* was demonstrated. Finally, the process of serialization and composition of AOT data into the ELF shared object structure, and the shared library approach was described.

Chapter 5

Results & Evaluation

The evaluation is focused on comparing ELF shared objects with the existing implementation of the shared cache in the *wabtaot* compiler infrastructure. Two *wabtaot* executables are generated, where one uses the ELF shared object as the storage container and the other uses the shared cache as a storage container. Both the *wabtaot* executables share almost everything identically, except the code storage container. Therefore, any difference in evaluation metrics is attributed to the code storage containers. Both runtimes are evaluated based on metrics like execution speed, memory usage, I/O utilization, and their sharing and concurrent execution capabilities.

5.1 Experimental Setup and Benchmarks

For all experiments, an isolated Linux benchmarking machine, which runs only one job at a time is used. The configuration of the Linux benchmarking machine is follows:

- Processor: Intel(R) Core/12 i7-8700 CPU 3.20GHz
- RAM: 32GB @ 2666 MHz

- OS: Ubuntu 20.04 (Focal Fossa)

Two suites are considered: WebAssembly bytecode test suite [7] used by wasmjit-omr runtime and the PolyBenchC benchmark [47]. All programs written in the WebAssembly bytecode test suite are in WebAssembly text format (WAT). These programs are converted to WebAssembly modules (.wasm) using the *wat2wasm* WABT tool. The second benchmark is PolyBenchC, version 4.2, which is written in C [36, 42]. The compute-intensive programs in the PolyBenchC benchmark perform numerical procedures like matrix multiplication and LU decomposition [47]. First, the PolyBenchC programs are converted to WebAssembly modules using Emscripten compiler, version 1.39 [17]. While compiling the PolyBenchC kernel using the Emscripten, the `ALLOW_MEMORY_GROWTH` flag is set to 1 to allow memory allocation whenever necessary. PolyBenchC can be compiled for multiple datasets like extra-large, large, medium, small and mini. The size of the dataset affects the execution time of each module. For example, all programs in the mini and the small datasets take around 4 seconds to execute. It is determined that the large dataset contains both short-running as well as long-running programs, so the large dataset is chosen for this evaluation.

5.2 Execution Time

The first metric to compare the shared cache and the ELF shared object is wabtaot execution time in compile and load runs. The perf 5.4.65 tool [2] is used to record the execution time of each run. 100 runs of each WebAssembly module mentioned in Table 5.1 are recorded, and the mean of the "elapsed time" statistic is reported. Overall, the standard deviation is always less than 1 percent of the mean execution time, per PolyBenchC program. Initially, a comparison using the bytecode coverage test suite is conducted to calculate the compile run execution time for modules that

Table 5.1: Comparison of the shared cache and the shared object execution time in **COMPILE RUNS** (in seconds).

	Modules	Shared Cache	Shared Object	% Difference
Bytecode Coverage	callindirect	0.17515	0.01963	−792.26
	reinterpret	0.17025	0.01469	−1058.95
	i64_bitwise	0.20047	0.04605	−335.33
	f64_comparison	0.27762	0.12253	−126.57
	get_set_tee_local	0.28089	0.12799	−119.46
	i32_arithmetic	0.30721	0.15456	−98.76
Short-Running PolyBenchC	atax	3.93317	3.78079	−4.03
	bicg	3.95617	3.81266	−3.76
	durbin	3.87229	3.72907	−3.84
	gemver	4.08574	3.94824	−3.48
	jacobi-1d	3.83939	3.70567	−3.60
	mvt	3.96987	3.83434	−3.53
	trisolv	3.86675	3.72646	−3.76
Long-Running PolyBenchC	correlation	10.63241	10.48523	−1.4
	3mm	16.76755	16.86126	0.55
	adi	28.49008	28.44438	−0.16
	seidel-2d	40.30751	40.11393	−0.48
	cholesky	69.14053	69.75818	0.88
	ludcmp	76.62393	77.47249	1.09
	floyd-warshall	90.90098	90.66453	−0.26

Note: A negative number indicates the percentage by which the ELF shared object approach is better than the shared cache.

take less than one second to execute. In Table 5.1, it can be observed that in compile runs the ELF shared object outperforms the shared cache, however, as seen in Figure 5.2, load runs of the shared cache are better than the shared object. The underlying cause of the overhead in compile runs is initialization and creation of the shared cache. On the other hand, the slowness in load run is attributed to the dlopen API, which loads and fetches the address of each symbol.

Furthermore, the comparison between the shared cache and the ELF shared object running PolyBenchC programs is carried out. After these programs are converted into WebAssembly modules, they are divided into short-running (3-4 secs.) and long-running (10+ secs.) categories, based on their execution time for compile runs. In the short-running PolyBenchC modules, it is observed that execution using the

Table 5.2: Comparison of the shared cache and the ELF shared object execution time in **LOAD RUNS** (in seconds).

	Modules	**Shared Cache**	**Shared Object**	**% Difference**
Bytecode Coverage	callindirect	0.00467	0.00507	3.83
	reinterpret	0.00432	0.00502	13.94
	i64_bitwise	0.00767	0.00815	5.88
	f64_comparison	0.02473	0.02517	1.74
	get_set_tee_local	0.02646	0.02737	3.32
	i32_arithmetic	0.00844	0.00913	7.55
Short-Running PolyBenchC	atax	0.07577	0.07748	2.2
	bicg	0.07525	0.07653	1.67
	durbin	0.03519	0.03622	2.84
	gemver	0.12521	0.1265	1.01
	jacobi-1d	0.02541	0.0261	2.68
	mvt	0.09371	0.09589	2.27
	trisolv	0.04634	0.04781	3.07
Long-Running PolyBenchC	correlation	6.61551	6.59073	−0.37
	3mm	12.85777	13.05699	1.52
	adi	24.49316	24.49856	0.02
	seidel-2d	36.41093	36.39239	0.04
	cholesky	64.57235	64.60461	0.04
	ludcmp	73.51225	73.01414	−0.68
	floyd-warshall	87.10911	87.08735	−0.02

Note: A negative number indicates the percentage by which the ELF shared object approach is better than the shared cache.

Table 5.3: Comparison of the shared cache and the ELF shared "library" execution time in **LOAD RUNS** (in seconds).

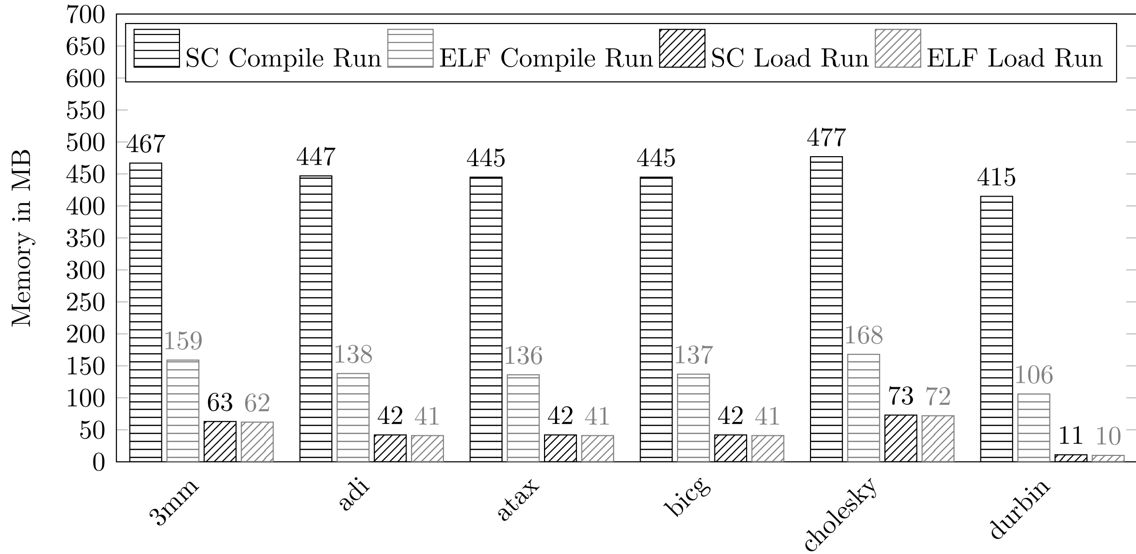
	Modules	Shared Cache	Shared Library	% Difference
Bytecode Coverage	callindirect	0.012772	0.0114613	−11.44
	reinterpret	0.012269	0.0105654	−16.12
	i64_bitwise	0.015354	0.012348	−24.34
	f64_comparison	0.02897	0.025582	−13.24
	get_set_tee_local	0.031969	0.029061	−10.01
	i32_arithmetic	0.029588	0.025516	−15.96

Note: A negative number indicates the percentage by which the ELF shared object approach is better than the shared cache.

shared cache is around 4% percent slower than execution using the ELF shared object in compile runs. However, in load runs the execution in the shared cache is slightly faster (2-3%) than the ELF shared object. In long-running modules, all compile runs of both storage containers indicate a difference of up to 2%. Similarly, all load runs also indicate a small gap of up to 2%. Overall, in all PolyBenchC modules, the variance does not cross 5%, which implies using the ELF shared object approach does not introduce major overhead and is a minor trade-off between compilation and loading performance, with respect to execution time.

Another experiment that we conducted involved creating a big shared library and subsequently loading from it. For this, 50 WebAssembly bytecode programs are compiled and a wasmaot.so file is generated. To increase the code/file size and number of methods, all functions from `f64_comparison.wasm` are replicated over 30 times with different names. This resulted in a shared library (wasmaot.so) containing over 5000 methods of size 657 KB. Then, a shared library inclusive approach is used to "only" load the modules mentioned in Table 5.3. 1000 load runs of each mentioned module are executed and report an average standard deviation of 1%. Previously, when the shared object exclusive method was used, load runs of the shared cache were superior to that of the shared object. However, as seen in Table 5.3, loading and executing a module from a huge ELF shared library is better than the shared cache. Therefore, it was inferred that an increase in code size and number of

Figure 5.1: Physical Memory Consumption of PolyBenchC Modules

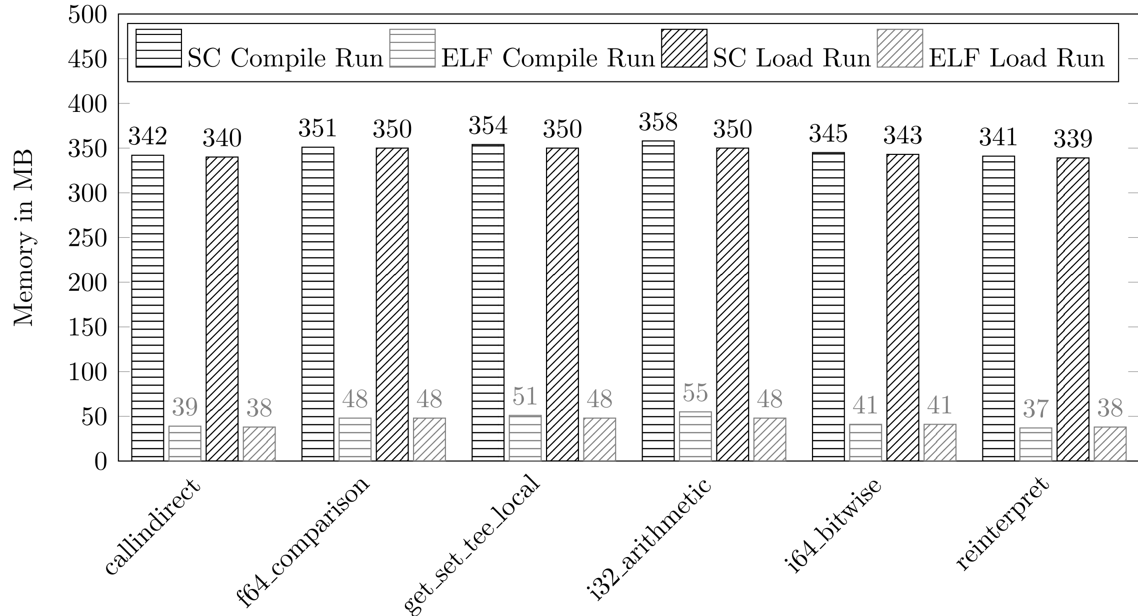


symbols in a code container can affect the start-up timings of the wabtaot runtime. This experiment is not conducted using PolyBenchC because PolyBenchC modules generated by the Emcripten compiler produced symbols that created conflict in the wabtaot runtime, when executed together.

5.3 Memory Usage

The memory consumption analysis is carried out using bytecode coverage tests and the PolyBenchC suite. The physical memory consumption is calculated using the GNU time (version 1.7) [1] command's *maximum resident set size* (*RSS*) statistic. RSS indicates the amount of memory a process is using from physical memory (RAM). Figure 5.1 displays the maximum RSS consumed by each run. When comparing the ELF shared object to the shared cache, it is observed that the bar corresponding to the shared cache compile runs heavily dominates the graph. This high usage is attributed to memory mapping a cache of a pre-specified size. In this case, the shared cache size is 300 MB, therefore the size-difference of approximately 300 MB. The shared cache size can be configured in the code, but

Figure 5.2: Virtual Memory Consumption of WebAssembly Modules



cannot be changed dynamically. In contrast, the runtime that uses the ELF shared object, demonstrates dynamic behavior, i.e., varies based on the AOT code and data stored in the shared object. Therefore, the physical memory consumption of the ELF shared object is less, as compared to the shared cache in compile runs, as well as in load runs. As seen in Figure 5.1, in load runs the physical memory consumption of ELF shared objects is 2.5% less than the shared cache, on average. To calculate the virtual memory utilization, the valgrind massif tool [20], version 3.15.0 is used. The valgrind massif tool is a heap profiler, which measures the heap memory utilized by an application. The flags *–trace-children=yes*, *–peak-inaccuracy*=0.5 and *–pages-as-heap=yes* are used to find the peak memory utilization of each wabtaot module. On executing the valgrind command for a module on multiple occasions, it is noticed that the peak is the same in all runs. Hence, the peak is reported from a single compile and load run of each module. In Figure 5.2, memory consumption of bytecode coverage test modules during compile and load runs using both containers is showcased. The compile runs and the load

runs of the shared cache consume more memory than the ELF shared object. As mentioned earlier, the cause of this is the mapping of a fixed-size cache.

The virtual memory utilization of each PolyBenchC module is analyzed. As these PolyBenchC modules are Emscripten-compiled, wabtaot needs to allocate over 2 GB for the module memory. Overall, in the shared cache compile runs, the peak memory consumption is about 2.83 GB for all the PolyBenchC modules. However, in compile runs of the ELF shared object, the virtual memory consumption is 2.52 GB for all PolyBenchC modules. This 12.7% gap is again attributed to the fixed-sized cache length. Similarly, in load runs, the shared cache touches the peak at 2.7 GB, while the ELF shared object touches the peak at 2.4 GB, thereby reporting a steady 13% difference. The ELF shared object only stores the necessary code and data in it, thereby consuming 300 MB less virtual memory. While usage of the runtime in memory-constrained environments is subject to the rest of the runtime being optimized for memory consumption, the ELF shared object is clearly a better candidate, since it produces lightweight containers for short-running programs.

5.4 File Size and I/O Performance

As mentioned earlier, the ELF-based approach has an option to generate an ELF shared object per compiled module. Whereas, all modules in the shared cache approach are compiled and stored in a single file, which can hold up to 300 MB of code and data. In this evaluation, it was noticed that all bytecode coverage test modules generate an ELF shared object that is no more than 25 KB. All PolyBenchC modules generate an ELF shared object that is no more than 140 KB. The shared cache file always occupies 300 MB of the file system regardless of how much code and data it stores. On that account, the ELF shared object can be highlighted as a lightweight container that generates a dynamic-sized object file.

The file I/O statistics of using the shared cache approach and of the ELF shared object approach are retrieved using the GNU time command. Using the time command, the results of the *File System Output* statistic, which indicates the number of file system outputs produced by a process are captured. The resulting number represents the total number of bytes written, divided by the block size [1]. In compile runs the shared cache reports around 600K outputs and the ELF shared object reports around 300 outputs for almost every run. This huge difference is attributed to the write operations that take place during the initialization and creation of the shared cache. The other metric that is captured is the "File System Input". This metric represents the number of bytes read from the file system. In all compile/load runs of both storage containers, it can be noticed that the file system input is always zero. This zero does not indicate zero reads from the file system, but is regarded as an inability of the time command to measure the reads [1].

5.5 Sharing and Concurrency

An underlying reason to develop a `dlopen` compatible ELF shared object is to have access to the features provided by the well-supported API. For example, the `dlopen` API provides benefits like sharing and concurrent execution. The ELF shared object can be accessed by multiple instances of wabtaot. As seen in Table 5.4, eight modules are selected: three from the bytecode coverage test suite, two from short-running PolyBenchC modules and the other three from PolyBenchC modules. These modules are randomly selected from each category for demonstration of concurrent execution in all three categories. To demonstrate concurrent execution, first, a compile run is carried out and then 10 concurrent load runs are carried out by generating background jobs. This experiment is repeated for 10 times on each module and the standard deviation is no more than 10%. As

Table 5.4: Comparison of the shared cache and the ELF shared object execution time for 10 CONCURRENT **LOAD RUNS** (in seconds).

Modules	Shared Cache	Shared Object	% Difference
callindirect	0.05609	0.00926	−505.76
f64_comparison	0.22405	0.05452	−310.94
i32_arithmetic	0.22519	0.03728	−504.11
atax	0.76624	0.14416	−431.54
bicg	0.75574	0.13774	−448.68
correlation	66.67143	14.26851	−367.26
seidel-2d	363.91803	67.24310	−441.20
floyd-warshall	870.85507	170.78778	−409.90

Note: A negative number indicates the percentage by which the ELF shared object approach is better than the shared cache.

seen in Table 5.4, the total execution time with ELF shared object as a container is around 400% better than the total execution time of the shared cache approach. The primary reason behind this huge execution time gap is the lack of synchronous execution support while using the shared cache as a container. This implies that each instance of the runtime that uses the shared cache waits for the other to complete. However, this is not the case for the runtimes using an ELF shared object, since they rely on the dlopen API for which the OS handles concurrency.

Chapter 6

Conclusion and Future Work

In this research, a better lightweight AOT code storage option i.e., the ELF-based shared object is presented. The language-agnostic *aotelf* design in Eclipse OMR will enable other runtimes to store their AOT data in an Eclipse OMR AOT-specific ELF shared object. New classes and methods introduced in the enhanced ELF generation module will allow language-runtime developers to write an adaptable ELF object as per their needs.

In this research, the ELF-based shared object approach is compared to the existing wabtaot's example implementation of the shared cache. In analysis, the trade-offs observed in the execution speed are minimal. Conclusively, the benefits of using the ELF shared object as a container are code sharing, smaller memory consumption, compact object files, reduced I/O utilization and access to features handled by the OS when the `dlopen` API is used. Therefore, the ELF shared object is characterized as a lightweight container, which is suitable for resource-constrained environments like embedded devices and IoT systems. The key highlight of the ELF shared object was its ability to execute synchronously when being consumed by multiple instances of a runtime. The concurrency experiment demonstrates that the ELF shared object approach allows accessing benefits of the shared object

support while retaining Eclipse OMR features. Other than the benefits of ELF shared objects, this thesis also presented a way to achieve multiple inheritance using the extensible classes.

In the future, the ELF shared object generator in Eclipse OMR can be improved by introducing the GNU hashing logic, instead of system-V style hashing. This will potentially enhance the symbol look-up process in the shared object, thereby reducing the loading time. Another major advancement to the ELF generation infrastructure can be achieved by introducing sections like *.rela.dyn*, *.got*, *.plt* and *got.plt*. These sections store the information that is needed to resolve the relocations at runtime using the dynamic linker/loader (ld.so) [49]. The dynamic linker/loader can resolve the relocation instead of the AOT relocation infrastructure. Furthermore, the dynamic linker/loader can eschew the loading of the AOT data into the Eclipse OMR code cache. Some might argue that a slower compile run is acceptable, but load runs always need to be better. Therefore, more investigation can be done to accelerate the load run using the ELF shared objects.

References

[1] time(1) — Linux manual page, 2019. URL: https://man7.org/linux/man-pages/man1/time.1.html.

[2] perf-stat(1) — Linux manual page, 2021. URL: https://man7.org/linux/man-pages/man1/perf-stat.1.html.

[3] The WebAssembly System Interface (WASI), 2021. URL: https://wasi.dev/.

[4] Syrus Akbary. A WebAssembly Compiler tale, 2019. URL: https://medium.com/wasmer/a-webassembly-compiler-tale-9ef37aa3b537.

[5] Syrus Akbary. Wasmer 1.0, 2021. URL: https://medium.com/wasmer/wasmer-1-0-3f86ca18c043.

[6] John Aycock. A Brief History of Just-in-Time. *ACM Comput. Surv.*, 35(2):97–113, June 2003. doi:10.1145/857076.857077.

[7] Leonardo Banderali. wasmjit-omr, 2019. URL: https://github.com/wasmjit-omr/wasmjit-omr.

[8] N. Barkakati. *Red Hat Fedora Linux Secrets.* John Wiley & Sons, 1st edition, 2005.

[9] Michael Barr and Anthony Massa. Programming Embedded Systems: With C and GNU Development Tools. 2011.

[10] IBM Corporation Ben Corrie, Hang Shao. Class sharing in eclipse openj9, 2018. URL: https://developer.ibm.com/tutorials/j-class-sharing-openj9/.

[11] D. Bhattacharya, K. B. Kent, E. Aubanel, D. Heidinga, P. Shipton, and A. Micic. Improving the performance of JVM startup using the shared class cache. In *2017 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, pages 1–6, Aug 2017. doi:10.1109/PACRIM.2017.8121911.

[12] c0ntex. How to hijack the Global Offset Table with pointers for root shells. URL: http://www.infosecwriters.com/text_resources/pdf/GOT_Hijack.pdf.

[13] Kusuma chalasani. Startup performance of tomcat in docker,eclipse openj9 blog, 2018. URL: https://blog.openj9.org/2020/05/18/startup-performance-of-tomcat-in-docker/.

[14] The GNU Compiler Collection and GNU Toolchain. ELFUTILS. URL: https://sourceware.org/elfutils/.

[15] The GNU Compiler Collection and GNU Toolchain. GNU Binutils, 2008. URL: https://www.gnu.org/software/binutils/.

[16] TIS Committee. Tool Interface Standard (TIS), Executable and Linking Format (ELF) Specification , 1995. URL: https://refspecs.linuxfoundation.org/.

[17] Emscripten Contributors. Emscripten Documentation, 2015. URL: https://emscripten.org/docs/index.html.

[18] IBM Corporation. Class data sharing, 2020. URL: https://www.ibm.com/support/knowledgecenter/SSYKE2_8.0.0/com.ibm.java.vm.80.doc/docs/shrc.html#shrc.

[19] IBM Corporation. Eclipse openj9 virtual machine, 2020. URL: https://www.ibm.com/support/knowledgecenter/SSYKE2_8.0.0/com.ibm.java.80.doc/user/java_jvm.html.

[20] Valgrind™ Developers. Massif: a heap profiler, 2021. URL: https://www.valgrind.org/docs/manual/ms-manual.html.

[21] Damian Diago D'monte, Georgiy Krylov, Daryl Maier, Gerhard W. Dueck, and Kenneth B. Kent. An ELF-Based Storage Option for the Eclipse OMR Ahead-of-Time Compiler. In *Proceedings of the 30th Annual International Conference on Computer Science and Software Engineering*, page 173–178, USA, 2020. IBM Corp.

[22] Damian Diago D'monte, Georgiy Krylov, Younes Manton, Gerhard W. Dueck, and Kenneth B. Kent. A Lightweight Code Storage Container for the Eclipse OMR Ahead-of-Time Compiler. In *Proceedings of the 31st Annual International Conference on Computer Science and Software Engineering*, CASCON '21, Canada, 2021. IBM Corp.

[23] JDK 10 Documentation. Ahead-of-Time Compilation, Java HotSpot Virtual Machine Performance Enhancements, 2018. URL: https://docs.oracle.com/javase/10/vm/java-hotspot-virtual-machine-performance-enhancements.htm#JSJVM-GUID-F33D8BD0-5C4A-4CE8-8259-FD9D73C7C7C6.

[24] Ulrich Drepper. How To Write Shared Libraries. Technical report, Technische Universität Darmstadt, 2011.

[25] Karel Driesen and Urs Hölzle. The Direct Cost of Virtual Function Calls in C++. *SIGPLAN Not.*, 31(10):306–323, October 1996. doi:10.1145/236338.236369.

[26] Irwin D'Souza. Ahead of time compilation: Relocation, Oct 2018. URL: https://blog.openj9.org/2018/10/26/ahead-of-time-compilation-relocation/.

[27] Irwin D'Souza. Intro to ahead of time compilation, Oct 2018. URL: https://blog.openj9.org/2018/10/10/intro-to-ahead-of-time-compilation/.

[28] Sue Chaplain Eclipse foundation, Dan Heidinga. Eclipse openj9; not just any java virtual machine, 2018. URL: https://www.eclipse.org/community/eclipse_newsletter/2018/april/openj9.php.

[29] Alessandro Di Federico, Amat Cama, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. How the ELF ruined christmas. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 643–658, Washington, D.C., August 2015. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/di-frederico.

[30] IBM/UNB Centre for Advanced Studies Atlantic. wabtaot, 2020. URL: https://github.com/CAS-Atlantic/wabtaot.

[31] Eclipse foundation. Omr project proposal, 2016. URL: https://projects.eclipse.org/proposals/omr.

[32] Eclipse foundation. Elfgenerator, eclipse omr. github, jun 2018. URL: https://github.com/eclipse/omr/blob/master/doc/compiler/runtime/ELFGenerator.md.

[33] Eclispe Foundation. Eclipse OMR, Issue No 2799, 2018. URL: https://github.com/eclipse/omr/issues/2799.

[34] Eclispe Foundation. Eclipse OMR, Issue No 4566 - AOT module for OMR compiler , 2019. URL: https://github.com/eclipse/omr/issues/4556.

[35] W3C Community Group. Should we use elf as a container format?, issue 74. GitHub, 2015. URL: https://github.com/WebAssembly/design/issues/74.

[36] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the Web up to Speed with WebAssembly. *SIGPLAN Not.*, 52(6):185–200, June 2017. doi:10.1145/3140587.3062363.

[37] Ludovic Henry. AOT Compilation in HotSpot: Introduction, oct 2019. URL: https://devblogs.microsoft.com/java/aot-compilation-in-hotspot-introduction/.

[38] Pat Hickey. Announcing Lucet: Fastly's native WebAssembly compiler and runtime, 2019. URL: https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime.

[39] Eclipse OMR IBM Corporation. Produce an object file with compiler/jitbuilder, issue 1170. GitHub, 2017. URL: https://github.com/eclipse/omr/issues/1170.

[40] Bill Budge (WebAssembly internals). Code caching for WebAssembly developers, 2019. URL: https://v8.dev/blog/wasm-code-caching.

[41] WebAssembly internals. What is V8, 2019. URL: https://v8.dev/.

[42] Petar Jelenkovic. Ahead-of-Time Compilation of WebAssembly using Eclipse OMR, 2020.

[43] Vladimir Kozlov. JEP 295: Ahead-of-Time Compilation, OpenJDK, may 2018. URL: https://openjdk.java.net/jeps/295.

[44] Georgiy Krylov, Gerhard W. Dueck, Kenneth B. Kent, Daryl Maier, and Irwin D'Souza. Ahead-of-time compilation in OMR: overview and first steps. In

Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering, CASCON 2019, Markham, Ontario, Canada, November 4-6, 2019, pages 299–304. ACM, 2019. URL: `https://dl.acm.org/doi/abs/10.5555/3370272.3370305`.

[45] Georgiy Krylov, Petar Jelenkovic, Kenneth B. Kent, , Daryl Maier, Gerhard W. Dueck, Mark Thom, and Younes Manton. Ahead-of-Time Compilation in Eclipse OMR on Example of WebAssembly. In *Proceedings of the 31st Annual International Conference on Computer Science and Software Engineering*, CASCON '21, Canada, 2021. IBM Corp.

[46] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1999.

[47] the Ohio State University Louis-Noel Pouchet, Tomofumi Yuki. PolyBenchC: the polyhedral benchmark suite, 2020. URL: `http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/polybench.html`.

[48] Linux Programmer's Manual. *dlopen(3)*, sep 2017. URL: `https://man7.org/linux/man-pages/man3/dlopen.3.html`.

[49] Linux Programmer's Manual. *ld.so, ld-linux.so - dynamic linker/loader*, aug 2019. URL: `https://www.man7.org/linux/man-pages/man8/ld.so.8.html`.

[50] Samer AL Masri. Static Versus Dynamic Polymorphism When Implementing Variability in C++, 2018.

[51] Samer Al Masri, Nazim Uddin Bhuiyan, Sarah Nadi, and Matthew Gaudet. Software Variability through C++ Static Polymorphism: A Case Study of Challenges and Open Problems in Eclipse OMR. In *Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering*, CASCON '17, page 285–291, USA, 2017. IBM Corp.

[52] Wolfgang Mauerer. *Professional Linux Kernel Architecture*. Wiley Publishing, Inc., 1st edition, may 2008.

[53] Microsoft. Windows Subsystem for Linux Overview, april 2016. URL: `https://docs.microsoft.com/en-us/archive/blogs/wsl/windows-subsystem-for-linux-overview`.

[54] Santa Cruz Operation. System V Application Binary Interface, jun 2013. URL: `http://www.sco.com/developers/gabi/latest/contents.html`.

[55] PaX-Team. PaX address space layout randomization (ASLR), 2003. URL: `http://pax.grsecurity.net/docs/aslr.txt`.

[56] Leon Presser and John R. White. Linkers and loaders. *ACM Comput. Surv.*, 4(3):149–167, September 1972. URL: `http://doi.acm.org/10.1145/356603.356605`, `doi:10.1145/356603.356605`.

[57] Android Open Source Project. Android Runtime (ART) and Dalvik, 2020. URL: `https://source.android.com/devices/tech/dalvik`.

[58] Android Open Source Project. Implementing ART Just-In-Time (JIT) Compiler, 2020. URL: `https://source.android.com/devices/tech/dalvik/jit-compiler`.

[59] Bytecode Alliance project. The Wasmtime guide , 2019. URL: `https://medium.com/wasmer/a-webassembly-compiler-tale-9ef37aa3b537`.

[60] Mono Project. Ahead of Time Compilation (AOT), The Mono Runtime, 2016. URL: `https://www.mono-project.com/docs/advanced/runtime/docs/aot/`.

[61] Mono Project. AOT, 2020. URL: `https://www.mono-project.com/docs/advanced/aot/`.

[62] Adam Richard, Lai Nguyen, Peter Shipton, Kenneth B. Kent, Azden Bierbrauer, Konstantin Nasartschuk, and Marcel Dombrowski. Inter-JVM sharing. *Software: Practice and Experience*, 46(9):1285–1296, 2016. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2379, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2379, doi:https://doi.org/10.1002/spe.2379.

[63] Ignacio Sanmillan. Elf sections and segments, 2018. URL: https://www.intezer.com/blog/research/executable-linkable-format-101-part1-sections-segments/.

[64] Huzaifa Sidhpurwala. "hardening elf binaries using relocation read-only (relro)", 2019. URL: https://www.redhat.com/en/blog/hardening-elf-binaries-using-relocation-read-only-relro.

[65] Azul Systems. ReadyNow! plus Compile Stashing, Technology White Paper, 2019. URL: https://go.azul.com/readynow-plus-compile-stashing-0.

[66] Azul Systems. Falcon: Improving Java Performance through Better Compilation, Technology White Paper, 2020. URL: http://go.azul.com/falcon.

[67] Azul Systems. Use Compile Stashing, Using the Zing Virtual Machine, 2020. URL: https://docs.azul.com/zing/UseZVM_CompileStashing_Overview.htm.

[68] Mark Thom, Gerhard W. Dueck, Kenneth B. Kent, and Daryl Maier. A survey of ahead-of-time technologies in dynamic language environments. In *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering, CASCON 2018, Markham, Ontario, Canada, October 29-31, 2018.*, pages 275–281, 2018.

[69] Mark Thom, Gerhard W. Dueck, Kenneth B. Kent, and Daryl Maier. Pervasive Sharing of Language Runtimes in Eclipse OMR, Internal Technical Report. Technical report, University of New Brunswick, Dec 2019.

[70] WebAssembly. WABT: The WebAssembly Binary Toolkit, 2019. URL: `https://github.com/WebAssembly/wabt`.

[71] Ian Wienand. PLT and GOT - the key to code sharing and dynamic libraries, May 2011. URL: `https://www.technovelty.org/linux/plt-and-got-the-key-to-code-sharing-and-dynamic-libraries.html`.

[72] T. Xinyu, Z. Changyou, L. Chen, K. Aourra, and L. YuanZhang. A Code Self-Relocation Method for Embedded System. In *2017 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*, volume 1, pages 688–691, 2017.

[73] Peilin Ye. Understanding _dl_runtime_resolve(). Technical report, Dec 2019. URL: `https://ypl.coffee/dl-resolve/`.

[74] Chao Zhang, Lei Duan, Tao Wei, and Wei Zou. SecGOT: Secure Global Offset Tables in ELF Executables. In *Proceedings of the 2nd International Conference on Computer Science and Electronics Engineering (ICCSEE 2013)*, pages 995–998. Atlantis Press, 2013/03. URL: `https://doi.org/10.2991/iccsee.2013.250`, `doi:https://doi.org/10.2991/iccsee.2013.250`.

Vita

Candidate's full name: Damian Diago D'monte

University attended (with dates and degrees obtained): Bachelor of Engineering (Computer), Fr. Conceicao Rodrigues College of Engineering, University of Mumbai, 2013-2016

Publications:

Damian Diago D'monte, Georgiy Krylov, Younes Manton, Gerhard W. Dueck, and Kenneth B. Kent. 2021. A Lightweight Code Storage Container for the Eclipse OMR Ahead-of-Time Compiler. Proceedings of the 31st Annual International Conference on Computer Science and Software Engineering (CASCON), Toronto.

Damian Diago D'monte, Georgiy Krylov, Daryl Maier, Gerhard W. Dueck, and Kenneth B. Kent. 2020. Position Paper, An ELF-based storage option for the eclipse OMR ahead-of-time compiler. Proceedings of the 30th Annual International Conference on Computer Science and Software Engineering (CASCON), Toronto, 173–178.

Conference Presentations:

Damian Diago D'monte, Georgiy Krylov, Daryl Maier, Gerhard W. Dueck, and Kenneth B. Kent. 2020. Poster, Persistent Storage for a WebAssembly Ahead of Time Compiler using Eclipse OMR. At the 30th Annual International Conference on Computer Science and Software Engineering (CASCON), Toronto.