BACHELOR'S THESIS

KONRAD WINSLOW

Dynamic Optimization of WebAssembly on low-resource Devices

Department of Computer Science and Mathematics
Munich University of Applied Sciences

Supervisor
Prof. Dr. Stefan Wallentowitz

December 2022 – version 1

ABSTRACT

The restrictions native, platform-specific programs put on an application's portability and compatibility are especially problematic for embedded devices, as different target platforms are very diverse. To address such challenges, a program may be interpreted, translating virtual instructions to machine instructions. However, this additional layer of abstraction usually comes with a penalty on execution speed due to the interpreter's overhead,

To improve the runtime performance of interpreted code, advanced runtimes perform optimizations on a program during execution, a strategy commonly referred to as just-in-time (JiT) compilation. In contrast to optimizations performed by compilers ahead of time, a virtual machine is not restricted to making provably correct assumptions about the program and can base compilation on speculations that may be invalidated.

A modern, standardized and open virtual instruction format known as WebAssembly (WASM), originally developed for web browsers, is finding wider usage in embedded systems, smart-, and IoT devices. Available WASM runtimes for embedded systems do currently not employ a complex optimization pipeline and perform at best static optimizations by employing preexisting JiT frameworks such as LLVM JiT.

This thesis evaluates the feasibility of employing a speculative, adaptive optimization framework for WASM applications on low-resource, embedded devices. For this, a WASM runtime was extended with profiling, optimization and de-optimization capabilities. The challenges of translating such strategies to our target hardware and implementation techniques are described. Measurements show that a dynamic optimization framework for speculative JiT compilation is feasible on different SoCs and likely to have a noticeable performance increase compared to interpretation.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

WASM WebAssembly

JiT Just in time

OSR On-stack replacement

PIC Polymorphic inline cache

TCO Tail-call optimization

wat WebAssembly text format

WASI WebAssembly System Interface

HAL Hardware abstraction layer

MCU Microcontroller

1 INTRODUCTION

1.1 MOTIVATION

Some IoT devices and embedded systems, like smart cards, load and execute user applications on demand, instead of being programmed ahead of time. For example, on devices supporting proprietary Java Card technology, interoperable java applets such as authentication clients can be loaded at runtime to extend core functionality. If not for a virtual machine interpreting an application, as is the case with Java Card, programs would have to be compiled for each native target platform, accounting for peripherals, extensions and different machine instruction sets.

WebAssembly offers a promising technology for virtualizing such applets: Its bytecode format can easily be mapped to machine instructions, and a variety of source languages, such as C, Rust and C++ can be compiled to WASM.

However, the computational power of embedded devices is low, worsening the negative impact the use of a virtual machine has on execution efficiency. To improve performance, an interpreted program can be optimized during runtime by just-in-time compilation. Such dynamic optimizations can use observations of the program to limit optimizations to frequent code sections or to perform speculative optimizations. JiT compilation drastically improves runtime performance, leading to better responsiveness and lower processor requirements. With future development, a universal WASM runtime containing a complex compilation pipeline, performing static and dynamic optimizations could enable applets to run efficiently on smart- and IoT devices in a platform-agnostic manner.

1.2 GOAL

Several WebAssembly runtimes for non-web environments currently exist. These runtimes provide an interface for interaction between a WebAssembly application and the host, and can be integrated as a subsystem into more complex applications. Support for just-in-time compilation on popular WASM interpreters is in the early stages of development, but efforts are so far restricted to performing simple, static optimizations like register allocation.

Regarding other interpreted languages, advanced runtimes, like the HotSpot VM, GraalVM, luaJIT, V8 or SpiderMonkey, make wide use of adaptive, feedback-guided optimizations, due to which they can, in some cases, outperform ahead-of-time compilers that can't use live runtime feedback.

The execution efficiency of WebAssembly on embedded devices might similarly benefit when such feedback-guided optimizations are utilized. Not only do such dynamic optimizations have the potential to create better-performing code; they can better judge what code sections to spend compilation resources on. Challenges, restrictions and the potential performance impact of using dynamic, feedback-guided compilation on low-cost devices interpreting WebAssembly should be assessed and enable further work on the matter. For this, an existing WebAssembly interpreter is to be extended with dynamic optimization capabilities and serve as a basis for evaluation.

1.3 STRUCTURE OF THIS WORK

The contents of this thesis are structured as follows:
In Chapter 2, fundamentals of WebAssembly and optimizing interpreters are established. Chapter 3 discusses the approach and constraints of the experimental implementation as well as the technologies it is based on: In Section 3.1 the development- and profiling targets and the WebAssembly interpreter to be extended are introduced. A chronological outline of the development of dynamic optimization is given in Section 3.2.

Chapter 4 delineates the implemented extensions to the WASM interpreter and rationalizes adaptions made due to the target hardware or interpreter design. Measurements comparing the extended- to the baseline interpreter are listed in Chapter 5, which are used as the basis for predictions, discussion, and further work in Chapter 6.

2 BACKGROUND

2.1 WEBASSEMBLY

WebAssembly is a modern binary instruction set for a virtual stack-based machine [52]. It was originally designed as a fast, portable and small virtual code format for use in web browsers in conjunction with JavaScript. However, it is not restricted to the originally intended use case, as each environment WebAssembly is executed on can provide an API for interaction. For example, web browsers offer an execution environment with functionality for the interoperability of JavaScript with WASM [50]. Similarly, the WebAssembly System Interface (WASI) acts as a general abstraction layer for use in multiple non-web environments [20].

A WebAssembly application consists of a set of modules, all of which declare imported and exported items, such as functions, global variables, tables and memories. Using exports, a module can make functionality available to the runtime environment, also called the "embedder". Using imports, a module can interact with the embedder, for example by opening and writing files using corresponding imported functions. Because WebAssembly is a stack-based instruction architecture, operations consume their arguments from the stack and push their output to the stack. Note that the stack is implicit, and an execution environment might not use an actual stack for its implementation. Next to the stack, WASM modules have byte-addressable vectors called 'memories', tables containing values of a reference type, and variables that can each be modified and read by instructions. A notable attribute of WASM is that it runs inside a memory-isolated environment and a module's semantics and structure are validated for soundness. For instance, the signature of a function is checked at the call site to ensure it matches the expectation. Unlike most other bytecode formats, all of WASM's control instructions target well-defined structures, mirroring lexical control flow. Structured control instructions such as if/else, blocks or loops introduce labels to the module. Unstructured instructions like branches can only jump to labels currently in scope, and not to arbitrary bytecode positions. Labels are introduced based on scope: For blocks and if/else constructs the labels point to the end of each construct, while loops add labels to the scope that point to the beginning of the loop.

WebAssembly bytecode can be displayed in "WebAssembly text format" (wat) which formats a module's abstract syntax as s-expressions. We can convert between WASM bytecode and the text format, the latter being easier for humans to read and work with. A simple C-program with its WASM compilation result formatted in wat is shown in Listing 1.

```c
int add (int first, int second) {
        return first + second;
}
```

```wat
(module
  (type (;1;) (func (param i32
  ↪  i32) (result i32)))
  (func $add (type 1) (param i32
  ↪  i32) (result i32)
    local.get 1
    local.get 0
    i32.add)
  (memory (;0;) 2)
  (export "memory" (memory 0))
  (export "add" (func $add))
```

Listing 1: Add.c & Add.wasm

WebAssembly is not only designed to be platform-independent, but it is also independent of the source language it can be compiled from. This allows for a wide variety of source languages to support WASM compilation and interoperation [42]. WebAssembly's low-level nature and the absence of garbage collection or a memory model make it more popular as a target for low-level languages like C, rather than for feature-rich, high-level languages [22, 41].

There exist parallels between WebAssembly and Java Bytecode [29]. Both intermediate representations specify a stack-based virtual machine with the support of variables and other named memory locations and allow for different high-level languages to be compiled into the respective format [40, 48]. Either instruction format increases the portability of an application by transferring execution to a virtual machine, making it independent of the hardware platform. Even so, as WebAssembly is designed to be language-independent, while Java bytecode was originally developed for the Java programming language, WASM source languages are more diverse than those commonly used to generate Java bytecode [40, 49].

2.2 WEBASSEMBLY ON EMBEDDED SYSTEMS

The WebAssembly working group acknowledges that, although initially designed for the web, WASM can be executed on a diverse set of platforms, including embedded systems. This is achieved by WASM's platform-independent design; as WASM does not inherently contain any domain-specific bindings or instructions unless they are provided by the embedder [52].

2.2.1 *WebAssembly as an alternative to native code*

There are two main reasons to run a WASM application on an embedded system, as opposed to using native machine code.

Firstly, WebAssembly applications are implicitly portable between any platform that provides the required interface, while considerable effort goes into ensuring cross-platform compatibility for native programs [31, 43, 46]. When developing a native application with support for different target platforms, each of the targets' differences has to

be taken into account. It is common to develop a platform abstraction library alongside the application, which increases development costs [56]. Such a library is often called a hardware abstraction layer or HAL, which serves as middleware between the hard- and software. However, native programs including a HAL are not forward-compatible with regard to the set of supported platforms. Each time support for a new target is to be added, the abstraction layer has to be extended. In addition to the extension of the HAL itself, an application's code has to be recompiled to account for changes in the abstraction library. Furthermore, hardware abstraction layers are typically written in a single source language, such as C, and are thus not language-independent.

WebAssembly can solve problems of cross-platform development on embedded systems by only requiring a compatible embedder to be present for each supported target. This reduces development costs, as an application only has to target the already standardized WASM interface, and makes applications implicitly forward compatible to any platform which will host a compatible runtime. No recompilation is required to support new targets and WASM is independent of the source language used for development.

Secondly, WebAssembly enables the execution of multiple software components in secure, isolated environments. On embedded systems that execute more than one program concurrently and may have the ability to load arbitrary programs, security is a concern [37]. Applications should run in isolation so that they cannot access each other's memory or gather information from another process in an unprivileged manner. In addition, software faults in one application should not lead to the failure of other components. Usually, an operating system such as Linux offers isolation between different processes, but embedded devices may not offer all features required by typical operating systems, such as a memory management unit. Due to WASM's design, different modules are implicitly executed in isolation from each other. They cannot access memory in an unprivileged manner and the implementation of a WASM runtime does not require specialized hardware support [27].

2.2.2 *WebAssembly as an alternative to interpreted languages*

The advantages WebAssembly offers compared to native code can be achieved by many interpreted languages and instruction sets. High-level, interpreted languages have recently been gaining traction as an alternative to machine code on microcontrollers. A popular instance of such language is python, which can be run on "bare-metal" systems (which do not provide an operating system) using the MicroPython runtime [18].

There are drawbacks to the use of high-level, interpreted languages on embedded systems, which limits their use [43]. Languages that are interpreted on the target platform typically run a lot slower than a native program would. The reason for this is twofold: Interpretation of

virtual instructions always comes with computational overhead and languages such as python have dynamic language constructs and features that are difficult to implement on low-resource systems. For instance, python does not require the manual management of memory, so a garbage collector is required to be included in MicroPython [19]. In comparison, WebAssembly combines the portability of an interpreted, high-level language with a low-level design, enabling it to achieve better performance results [43]. For instance, WebAssembly outperforms a subset of JavaScript used as a compilation target for low-level programming languages: 'asm.js' [44]. Furthermore, WASM is more suited as a target for low-level programming languages, such as C, C++ or Rust, because of its simple and universal semantics [52]. This comes with the disadvantage of being a less suitable compilation target for high-level languages, that require runtime support for garbage collection, dynamic typing, or reflection.

As traditional interpreters only support the execution of a single language, other programming languages are incompatible. This restriction does not only limit the set of languages one can choose from for development; it also makes inter-operation between them difficult. Sometimes it is possible to transform the code of one high-level language into another, a process known as transpiling, but because of language-specific features, this is not always feasible. Owning to WebAssembly's design, multiple source languages can be combined into a single WebAssembly application and interact with each other in the same runtime.

2.2.3 *WebAssembly as an alternative to the JVM*

Unlike interpreters for high-level languages, the JVM and a WebAssembly embedder both execute intermediary instructions which a source program gets compiled into [29, 52]. In the context of embedded devices, WASM has certain advantages over Java bytecode [27]: more universal semantics, which are easier to map to machine instructions, and better support for low-level source languages [37, 52].

Regarding the first aspect, Java bytecode was originally designed as an intermediate format for the Java programming language. As such, many of its constructs and instructions are designed with Java's features in mind. For instance, almost all calls in Java are dynamically dispatched on an object, hence the JVM supports 'invokevirtual' and 'invokeinterface' instructions [29]. Specialized semantics make it more difficult to implement a runtime, especially on resource-constrained devices. WASM, with its low-level and platform-independent design, mostly supports instructions that are present on many native architectures. Furthermore, WASM's memory model consists of a stack, variables/locations, tables, and linear byte vectors, all of which require minimal overhead to translate to a machine's native memory model, which is typically also divided into a stack and byte-addressable memory.

The JVM supports features such as garbage collection, reflection, and dynamic dispatch, which are usually not used on embedded, bare-metal systems. As such, the JVM is a suitable target for high-level languages with features that closely match the JVM's semantics like Java or Kotlin. Low-level languages are better suited for representation as WASM, which performs no garbage collection and only hosts a simple memory model.

Notably, the Java platform "Java Card" tries to improve Java's compatibility with low-resource hardware and is being used on embedded devices, particularly on smart cards. Java Card is a proprietary [36] subset of the Java 2 programming language and is missing several language features from the full Java SE platform to allow for its execution on a resource-constrained environment. Java Card is licensed to hardware manufacturers on an OEM basis. However, Java Card does not attain the same source language universality as WASM can.

2.3 INTERPRETER OPTIMIZATIONS

Executing a program by interpreting its instructions, instead of directly executing them on the hardware, drastically decreases its execution speed, with early, interpreter-only Smalltalk-80 systems running benchmarks at an order of magnitude slower than equivalent C-programs [9].

Throughout the history of virtual machines, many techniques have been devised which optimize the execution speed, memory usage and code size of interpreted programs, allowing them to challenge the performance of native programs compiled with an optimizing compiler[4, 30].

One can distinguish between optimizations of the interpreter design, such as choosing more suited data structures, and the optimization of the interpreted program, such as by JiT or AoT compilation. The latter is relevant to this thesis and we refer to it using the term "optimizations".

The optimization of an interpreted program transforms it into a different form. We will refer to this transformation as compilation, even when we do not emit native code. For example, the program can be compiled into an intermediate representation. When compilation occurs before a program's execution takes place, it is classified as ahead-of-time (AoT) compilation, contrasting just-in-time (JiT) compilation, which is performed during runtime.

Strategies used in AoT compilation are outside of this thesis' scope. Note, however, that JiT compilers rely on many optimizations common to AoT compilation, like register allocation or loop unrolling.

2.3.1 *JiT-compilation Strategies*

Just-in-time compilation ultimately obtains a performance increase by applying similar optimizations as the ones used by AoT compil-

STRATEGY	SYSTEM	AUTHORS
Dynamic native-code generation	Smalltalk	Deutsch and Schiffman 1984[13]
Type prediction; customization; guarding	SELF	Chambers et al. 1989[8]
Polymorphic inline caches	SELF	Hölze et al. 1991[26]
Deoptimization; OSR	SELF	Hölze et al. 1992[24]
Adaptive/Tiered optimization; Feedback-guided optimization	SELF	Hölze and Ungar 1994[25]
Dynamic patching	Java	Cierniak et al. 2000[11]
Trace compilation	Dynamo; Java	Bala et al. 2000[7]; Gal et al. 2006[17]

Table 1: Historic developments of dynamic optimization

ers. However, instead of optimizing the entire program, JiT compilers can inspect the code's execution and decide on what parts to compile with how much effort. In addition, using execution feedback, optimization at runtime can be more aggressive than static, AoT optimization, which is restricted to transformations that provably maintain a program's semantics.

Modern interpreters can rival the efficiency of native machines by combining various JiT compilation strategies developed over the past 40 years. Table 1 summarizes relevant milestones, many of which were achieved through research on the Smalltalk, Self and Java programming languages. A JiT compiler requires two separate subsystems: A profiling and decision-making component, that gathers information about an executing program, makes decisions based on that information and propagates the profiling data to other components, and an "optimization" system that optimizes the executed program and integrates its results with the rest of the runtime.

2.3.1.1 *Dynamic optimization*

Dynamic optimization, performed by a dynamic compiler differs from standard, static optimization in being performed during the runtime of a program. This is the most general case of JiT compilation and the two terms may be used interchangeably.

Deutsch and Schiffman described a basic form of dynamic optimization called dynamic translation for an efficient Smalltalk-80 runtime [13]. Part of the idea was to translate virtual instructions to na-

tive machine code, which would be generated on demand by the control flow of a program, reducing the overall memory usage compared to eagerly compiling all code. A method would be compiled when first invoked, and then kept in a cache for future invocations.

The prototype-based SELF language, making heavy use of dynamic dispatch, improved upon Smalltalks-80's dynamic translation by inlining messages (method calls) and guessing the type of a message's receiver, which allows for further optimizations to be performed. The prediction of program attributes allows affected sections to be compiled in a more optimized manner by 'specializing' generated code to such attributes. Since the type prediction could prove incorrect in a dynamic language, a test and corresponding branch were inserted, preceding any specialized code, to fall back to an unoptimized version should the test fail. This technique is often referred to as "guarding" a section of code. Alternatively, control flow may be diverted based on the type of a variable evaluated by the guard, a strategy called message splitting by the authors. We refer to any dynamic optimizations that rely on assumptions that can be invalidated throughout a program's execution as "speculative". SELF's speculative type prediction relied on static information, like that the 'equals' operation is most often performed on two integers, and did not yet consider type feedback from the executing program.

2.3.1.2 *Feedback guided optimization*

Feedback-guided optimizations consider data gathered about the program at runtime and are inherently dynamic. Ahead-of-time compilers can similarly use prerecorded execution profiles, but these are not considered feedback, as the profile used during compilation is static in nature.

The third implementation of SELF extended the prior version by basing compilation decisions on dynamic feedback [25] and allowing for "adaptive optimization". With adaptive optimization, the responsiveness of a program is increased due to the use of fast, baseline compilation at the start of execution and using obtained feedback to recompile frequently executed code sections with further optimizations. The original work described using simple method invocation counters to identify "hot" methods. An invocation counter exceeding a threshold would trigger the more optimized recompilation of a currently executing function. To decide on what call frame to begin recompilation at, the size and call frequency of multiple methods on the call stack were observed. Switching to more complex, time-consuming compilation algorithms after recording profile data is common in modern runtimes such as V8 [6] or HotSpot [30], with three or even four tiers of compilation commonly existing. The initial, fast compiler of such architectures is often referred to as a "baseline" compiler.

With the new profiling capabilities, the third generation of SELF could base its type prediction on observed runtime feedback instead of the previously used static type information. This effectively allows

for "adaptive inlining", where the most frequent target of a dynamic call site gets inlined into a surrounding method.

2.3.1.3 *Deoptimization and recompilation*

Sometimes it is required to deoptimize or recompile a currently executing section of code, such as when an assumption proves to be invalid or when a more aggressive optimization should be performed.

Deoptimization was originally used for debugging optimized code in a SELF virtual machine [24] but has since been used to allow speculative optimization without having to compile a code path to take should a preceding guard fail. Two notable deoptimization techniques are on-stack replacement (OSR), as originally done in SELF, and dynamically patching code sections in a thread-safe manner. On-stack replacement is performed at "safepoints", at which a 1:1 mapping of optimized to unoptimized code exists: During OSR, an equivalent, unoptimized or recompiled call stack and runtime state are created for the active backtrace, depending on if deoptimization or reoptimization is to be performed. Afterward, the new functions replace all current activations on the stack, and execution continues where it was interrupted. SELF would use OSR to both deoptimize compiled code for debugging as well as to recompile a currently executing method when the invocation threshold was exceeded. Code patching, on the other hand, requires that a failure path is present at sites where an assumption may prove invalid, but instead of a guard, a direct jump to the optimized path is inserted. When an assumption is invalidated, the jump is patched and replaced with a comparison and a branch [11]. On-stack replacement has the added benefit of also allowing the system to replace functions with recompiled versions while they are still active.

Deoptimization can be triggered by a guard (synchronously) or by other components of the runtime system (asynchronously). For example, Java inlines virtual method calls without inserting a check for the receiver type should only one instance of the class exist. When a second compatible class is loaded at a later time, the now invalid section of code has to be "deoptimized" into a version compiled without the corresponding assumption.

2.3.1.4 *Trace compilation*

A recent approach to JiT compilation is called "trace compilation" [21]. The methods discussed so far operate on basic blocks or procedures of the program to optimize. Trace compilation combines the above strategies for dynamic optimization, but does not treat procedures as the unit of compilation. Instead, the program is instrumented to record a bounded trace of its control flow, starting at a "trace anchor", which is then compiled and optimized.

The trace usually passes through method or function boundaries and records control-flow information such as the invocation count of loops, taken branches, and call targets. Once completed, the trace

is compiled, with deoptimization points inserted where the actual control flow may differ from the recorded trace. The running program is typically being profiled to trigger trace compilation, similar to how SELF93 used invocation counts to trigger a recompilation.

2.3.1.5 *Profiling*

There exist different approaches to collecting runtime information for use in JiT-compilation [4].

One of the simplest forms of profiling is instrumentation - instrumentation inserts instructions into code that collect profiling data. For example, the described third implementation of SELF incremented invocation counters in method preludes [25]. Instrumentation can range from collecting simple metrics, such as method invocation counts, to complex ones, like recording every traversed basic block or tracing the entire control-flow path. Furthermore, instrumentation can be performed on native machine code as well as on interpreted code and may be dynamically enabled and disabled.

Another approach to profiling is 'sampling': when a program is sampled, its execution state is observed at certain points during execution for profile collection. For instance, the aforementioned SELF implementation samples the call stack when the counter threshold is exceeded to determine what base method to begin recompilation at. The sampling process can be triggered in different ways: In SELF93 a counter was used, but another common way is to sample at periodic intervals by using scheduling, hardware level interrupts, or other software-based techniques [5, 53].

Different profiling strategies can be combined to increase effectiveness. Since instrumentation can give detailed statistics but has a considerable overhead in optimized code, it can be dynamically added to an existing code section based on information obtained through sampling. Suganuma et al. describe a JVM that detects hot methods using a sampling profiler, instrumenting hot methods to collect detailed information when required [45].

If applicable, hardware performance monitors can also be used to gather profiling data, but such data sources limit the virtual machine to supporting platforms.

3 DESIGN

3.1 TECHNOLOGIES

3.1.1 *ESP32-C3*

To assess dynamic optimization on low-resource embedded devices, an ESP32-C3 was chosen as a hardware platform for profiling and development. Applications were built using a development framework that provides hardware abstraction of the platform and a runtime environment supporting dynamic heap allocation.

3.1.1.1 *Hardware platform*

The ESP32-C3 is a low-cost microcontroller SoC developed by Espressif systems [15]. The SoC supports 2.4GHz Wi-Fi (IEEE 802.11 b/g/n) and Bluetooth5 low-energy connectivity and aims to optimize power consumption, making it suitable for IoT applications. Up to 22 GPIO pins and various peripherals are accessible to the CPU.

The chip's processor is a single-core, 32-bit Risc-V CPU supporting the RV32IMC instruction set and operating at up to 160 MHz. Risc-V is an open instruction set that can be freely implemented and modified by manufacturers [38]. It takes a modular approach, containing a base instruction set that can be extended with standardized or custom extensions [39]. The ESP32-C3's processor supports the 'I' base integer instruction set, the 'M' standard extension for integer multiplication and division, and the 'C' standard for compressed, 16-bit instructions.

The processor follows a Harvard architectural approach and uses a separate instruction- and data bus (I-bus, D-bus). The CPU can access 400KB of internal SRAM, of which 384 KB are shared between I- and D-bus and 16KB are limited to the I-bus. In addition, 384KB of internal ROM are present, fully addressable by the instruction bus, and partially addressable by the D-bus as read-only memory. A maximum of 16MB of external flash memory is supported: up to 8MB of read-only data and 8MB of instructions. When dynamically compiling programs, the address space of the ESP32C3 has to be considered: accesses by the I-bus and D-bus are mapped to separate memory regions.

3.1.1.2 *Development Framework*

'Espressif's IoT Development Framework' (ESP-IDF) provides a software development kit, libraries and a board support package for the development of applications targeting a SoC of the ESP32 family [14].

ESP-IDF applications are scheduled as a task on the FreeRTOS[1] operating system, which enables multiprocessing, a libc implementation, and dynamic memory allocation. An ESP-IDF project consists of a set of components, each of which may depend on others. When building an executable, a project contains one 'main' component which implicitly depends on all other components and is added to the build target. On boot, the Risc-V processor starts execution of a first-stage, non-modifiable ROM bootloader, which transfers execution to the application. In a typical ESP-IDF executable, peripherals, the hardware, and a heap allocator are automatically initialized during startup to allow for the execution of high-level programs. Unless otherwise configured, the user application's entry point: '`void app_main();`' will be scheduled as a FreeRTOS task.

3.1.2 *MIMXRT1170-EVK*

NXP's i.MXRT1170 is a high-performance MCU designed to support real-time edge-computing-, automotive-, or industry applications [34]. The MCU has fewer resource restrictions than the ESP32-C3 and allows for the dynamic optimization of larger, more complex applications. For the evaluation of implemented changes, programs were profiled on the 'MIMXRT1170-EVK Evaluation Kit'.

3.1.2.1 *Hardware platform*

The platform supports multiprocessing by combining a 1GHz Arm Cortex-M7 and a 400MHz Arm Cortex-M4 processor. Both processors support Armv7-M Thumb instructions, which are 16- or 32-bit wide and 16-bit aligned, and both contain a floating point unit. The MCU contains 1MB of on-chip RAM, with additional 512 KB and 256 KB TCM memories of the two Cortex CPUs. Similar to the ESP32C3, the system follows a Harvard architecture, with separate instruction and data buses; however, instructions can also be fetched using the data bus, and memory is linearly mapped to addresses [2, 3]. The i.MXRT1170 MCU allows for various external devices and peripherals to be connected and for a maximum of 4GiB of 32-bit external SDRAM [35].

The MIMXRT117-EVK Evaluation Kit incorporates the i.MXRT1170 MCU, 512 Mbit external SDRAM memory, a total of over 2.5 Gbit of external Flash, and an onboard DAP-Link debugging interface.

3.1.3 *Development Framework*

NXP supports the i.MXRT1170 with their 'MCUXpresso' software development kit. The SDK comprises a board support package, peripheral drivers, CMSIS implementations, RTOS support, and 'newlib' libc bindings.

1 https://www.freertos.org/

To compile and run the interpreter on the target platform, MCUXpresso is configured to route stdout to a UART port and to set up a FreeRTOS task for the program, which is executed on the Cortex-M7 CPU. Code and data are written to the 512Mbit external SDRAM before execution

3.1.4 *Wasm3*

The WebAssembly interpreter Wasm3 targets a wide range of different platforms and can be used on different microcontrollers, including the ESP32-C3 and i.MXRT1170 [32]. Wasm3 relies on ESP-IDF and MCUXpresso to provide a runtime environment and heap allocation. Evaluating the use of WebAssembly on different environments, specifically on MCUs, is a particular goal for the development of Wasm3. Featurewise, the runtime currently only hosts an interpreter and does not perform any native code generation or feedback-directed optimization.

Most Finished WebAssembly proposals are supported by Wasm3, a notable exception being reference types[2], meaning the runtime does not support the manipulation of WASM tables at the time of writing. The interpreter hosts a WASI [20] compliant API, but it is not supported for all platforms, including the ESP32-C3.

3.1.4.1 *Architecture*

Wasm3 is written in the C programming language and makes use of the pre-processor for configuration and cross-compatibility. Wasm3 can either be built as a Windows, Linux or macOS command line utility or used as a library in other applications. Wasm3 bindings exist in popular languages such as C/C++, Rust, Zig, GoLang or Python3. To allow interaction with the host environment, applications can link native functions to imported functions of the interpreted WASM module.

Internally, Wasm3 can be subdivided into a parser, a compiler, an interpreter, and runtime data with utility functions. When using the library, a user application will interact with different parts of Wasm3 using exposed functions and handles. The basic description of each component and the interactions are described below:

- **User Application**
  An executable, either the Wasm3 command line utility or a user program, owns handles for Wasm3 modules, environments, runtimes and results. The WASM bytecode is provided by the user application for Wasm3 to parse and compile, and native functions can be linked to imported WebAssembly functions. Once initialized, the user application looks up an exported WASM function of a module and invokes it.

- **Runtime & Utility**
  Wasm3 provides interfaces to interact with internal data struc-

2 https://github.com/WebAssembly/reference-types

tures, for instance, loaded WebAssembly functions or modules. Some of these interfaces are available to user code, others are reserved for internal use. Such utilities provide logic for reallocating memory, setting identifiers, extracting result values, linking native functions and populating intermediate code pages.

- **Parser**
  An executable invokes the '`M3Result m3_ParseModule(...)`' function with a module's WASM bytecode to parse it and create required data structures. Wasm3 does not parse a module's code, as it is directly read by the compiler, but only the module's metadata, including its function types, exports, imports and memories.

- **Compiler**
  The runtime compiles WebAssembly functions when they are first invoked or looked up. The compilation consists of a linear pass over a function's WASM bytecode, throughout which intermediate code, or 'm3 code' is emitted. The compiler keeps track of WebAssembly's stack usage during compilation and maps its usage to locations in a 'register file' - an array of memory indexed using offsets. To account for stacking WebAssembly call frames, the base of the register file is incremented on function calls. Local values and return values are mapped to the bottom of a function's register file. A similar approach is taken for WASM blocks, whose specified semantics also involve stack input- and output values. As an optimization, the compiler may allocate a stack location in native registers instead of a location in memory. M3 operations typically exist in variants either consuming operands from slots within the register file or native registers, so that values can be spilled to memory if required.

3.1.4.2 *Execution*

The design of the interpreter follows a 'direct threaded' approach [28]: each intermediate instruction is handled by a corresponding native function. To encode a list of instructions, the respective function pointers are written to an array. During interpretation, a virtual program counter 'm3_Pc' points into the array and the pointees are sequentially invoked.

Each operation has the same signature and takes the virtual program counter, a pointer to the register file, as arguments. At the end of each operation, m3_Pc is incremented and the result of calling the next operation is returned. Due to tail-call optimization, this does not increase native stack usage, and the call gets compiled into an indirect jump. However, the operations for calling WebAssembly functions: `op_Call` and `op_Entry`, and the loop instruction `op_Loop` do cause allocation of their native stack frame because they cannot be tail-call optimized. This is because certain state, like the next instruction pointer, has to be preserved.

The layout of the intermediate code page, the native stack, and the register file during interpretation are displayed in Figure 1.

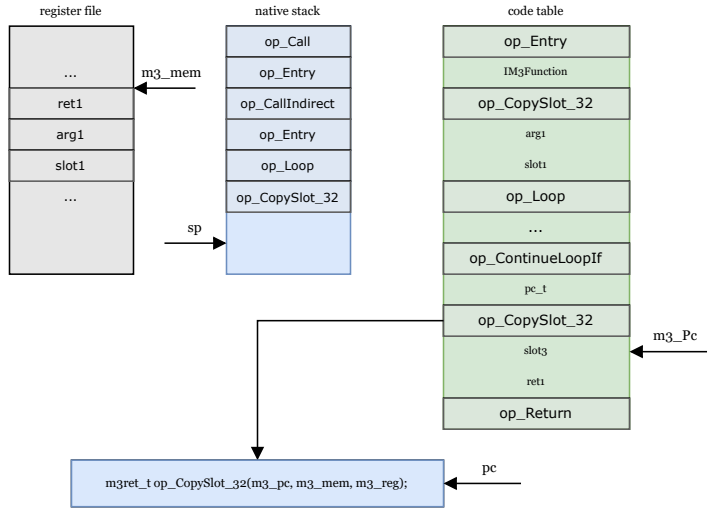


Figure 1: Wasm3 interpreter state during execution

The code page contains the instructions of the WebAssembly function that is currently executing, m3_Pc points to the current instruction. Some instructions take immediate arguments, like register file offsets, which follow the operation's function pointer in the code page. The native stack contains a call stack of operations requiring a persistent stack frame and the frame of the current operation. The next instruction will replace the topmost stack frame because of tail-call optimization. M3_mem points into the register file and acts as the memory base of a WASM function. Due to the direct threaded dispatch of Wasm3, the WASM call stack is mapped onto the native call stack, so no explicit data structure is required to enable function calls.

The source code of the currently executing instruction in Figure 1 is outlined in Listing 2. Note that `dst` and `src` are slots within the register file, defined by their offset which is given as an immediate, inline argument.

```
m3ret_t op_CopySlot_32(pc_t _pc, m3stack_t _sp, M3MemoryHeader * _mem,
↪  m3reg_t _r0)
{
    u32 * dst = (u32 *)(_sp + *((u32 *)_pc++));
    u32 * src = (u32 *)(_sp + *((u32 *)_pc++));

    * dst = * src;

    return (*_pc)(_pc + 1, _sp, _mem, _r0);
}
```

Listing 2: C source code for op_CopySlot_32

3.2 DYNAMIC OPTIMIZATION

The Wasm3 interpreter was extended with capabilities to perform dynamic, feedback-guided optimizations. Extensions made can be

subdivided into the two categories outlined in Section 2.3: profiling and optimization. Notable approaches within either category have been adjusted and implemented for the technologies described in Section 3.1 to form the framework of a dynamically optimizing interpreter.

The added framework performs adaptive, feedback-guided optimization but does not currently host a native compiler; instead, functions are recompiled into the same intermediate m3 code. This allows us to evaluate the impact of the systems governing a JiT compiler's decisions and which enable technologies such as recompilation and deoptimization introduced in Section 2.3.1.

3.2.1 *Profiling*

To allow for feedback-directed and speculative optimizations, Wasm3 has to collect runtime data about a WebAssembly module and trigger (re-)compilation that considers the collected profile.

Wasm3 already offered some profiling capabilities for the collection of debugging information, but these were unsuited for use in dynamic compilation. Much of the collected debugging information is not useful and would have to be collected for all tiers of optimization, because the interpreter did not allow for its fine-grained control.

To best assess the impacts and requirements of profiling WebAssembly bytecode on constrained devices, new profiling techniques were added to Wasm3 and existing ones were disabled. In total, the added profiling system captures the following traits through sampling and instrumentation: function invocation frequencies, backtraces, indirect call targets, and target invocation counts. The captured metrics are used to trigger recompilation, decide on the set of functions to recompile, direct recompilation, and to skip indirect function lookup routines.

3.2.2 *Optimization & Integration*

Once a hot function has been recompiled, execution should eventually switch to executing the new version. To ensure a switch takes place, all future calls have to be directed to the recompiled code by patching existing calls, changing metadata or inserting a stub. However, existing activations of a function remain unaffected by this - If a function on the active call stack contains a long-running loop, an interpreter may still spend a considerable amount of time executing unoptimized code, degrading performance. To prevent such cases, the active call frames can be replaced with activations of the recompiled code using on-stack replacement. While OSR for the re-optimization of code was introduced by Self-93 [25], later runtimes performing adaptive optimizations have not always followed the same approach. Instead, they simply waited for active functions to return, citing the complexities of implementing OSR and its increased resource requirements [12, 16]. To assess the implementation requirements and over-

head of an OSR mechanism, the extended version of Wasm3 employs the technique to replace any re-optimized functions on the call stack. When the profiling system triggers recompilation, the on-stack replacement mechanism ensures that the runtime will immediately switch to executing the newest version of code.

Smalltalk introduced a structure for capturing the invocation counts of dynamically dispatched calls: "(polymorphic) inline caches". These structures are unique because they fall into both categories of a profiling system as well as an optimization. Wasm3 was extended with polymorphic inline caches to both capture the frequency of taken indirect call edges, and to optimize WebAssembly programs with such runtime feedback.

Finally, old versions of recompiled code have to be freed and memory returned to the system. For this, garbage collection of code pages and ancillary metadata has been added to Wasm3.

4 IMPLEMENTATION

4.1 PROFILING

4.1.1 *Function invocation frequency*

The time spent by the interpreter in different WASM functions can be approximated using different approaches. Of the techniques outlined in Section 2.3, capturing invocation frequencies using instrumentation as described by [25] was chosen. Incrementing a value inside the prelude of a function is easy to implement and suited for use on low-resource devices that may not support the systems to periodically sample a program.

Hölzle and Ungar captured invocation counts of each method and let them decay exponentially over time; otherwise, every function would eventually trigger a recompilation. In SELF and JVM implementations, counters were often adjusted periodically through the use of scheduled context switches either between user-level, interpreter threads or operating system tasks. The period was typically shorter than the chosen half-time, and as such in each period, counters were divided by a floating-point constant between 1 and 2.

Because kernel- and user-level threads are not always viable on low-resource devices such as microcontrollers, a different decaying algorithm was implemented in Wasm3. Instead of periodically reducing every function's invocations, Wasm3's extension decreases counts lazily. When a function call causes a count to meet its threshold, the time passed since the last decay is calculated and the number is reduced accordingly. Afterward, the count is checked again, and if it still meets the threshold, a recompilation is triggered. Listing 3 shows the code responsible for calculating invocation rates at the entry of a function. To reduce the overhead of calculating invocation frequencies, especially in the absence of an FPU, the `decay()` function simply halves the value for each elapsed half-time by bit shifting, trading some precision for an increase in performance.

The number of function invocations is stored as an immediate argument to the `op_Entry` instruction.

4.1.2 *Target invocation frequency*

While the invocation frequency of a function is used to trigger recompilation on function entrance, the counts of every dispatched target per call site can be used to decide on optimizations such as inlining and to find a suitable root to begin recompilation at. Recording target invocation counts effectively gives weights to each edge in the

```
u32 *invoked = &immediate(u32);
(*invoked)++;
if (*invoked >= d_m3InvocationLimit) {
    if (*invoked == 0xF0000000) { // set initial timestamp
        *invoked = 0;
        function->lastDecay = m3GetClock();
        goto end;
    }
    decay(invoked, &function->lastDecay, m3GetClock()); // decay counter
    if (*invoked >= d_m3InvocationLimit) { // trigger recompilation
        return(m3Err_jitInvokeOverflow); // unwind stack
    }
}
```

Listing 3: Invocation-counting instrumentation preceding function entry

call graph of a WebAssembly program. Two cases were considered during implementation:

- **Static dispatch**
  In WebAssembly, the target of a 'call' instruction is well-defined by a function index into the module's function vector. To implement the capture of invocation counts for 'call' instructions, a new immediate argument was added to the `op_Call` instruction that is incremented on execution similarly to capturing the function invocation frequency.

- **Dynamic dispatch**
  WASM's 'call indirect' operation consumes an integer from the value stack. The operand indexes into a table containing function references, allowing for the dynamic dispatch of a target at runtime. Wasm3's extension allocates new counters per target lazily on their first invocation using a construct known as a 'Polymorphic Inline Cache', which is described in detail in Section 4.2.3 PICs serve for both profiling and optimization.

4.1.3 *Function Call-stacks*

When a recompilation is triggered by an invocation counter meeting a threshold, a function has to be chosen for recompilation. Typically, the function on top of the call stack is not the same function at which recompilation should begin, because then it may not be inlined at the call site.

To find a suitable root at which to start a recursive recompilation, an adjusted mechanism inspired by SELF was implemented. SELF-93 traversed the call stack, evaluating the number of times each method made calls to unoptimized or tiny code, to find a suitable recompilation target. As a prerequisite, a backtrace has to be captured by the profiling system, which is straightforward in an interpreter that stores the virtual call stack, but more difficult with directly threaded interpreters such as Wasm3.

Wasm3 maps the virtual WASM call stack to the native machine stack as a consequence of its architecture described in Section 3.1.4.2. Unless WebAssembly function activations are captured eagerly on their occurrence, which would result in performance degradation, the native stack has to be unwound to construct a WebAssembly backtrace. As a positive effect of the latter approach, performance penalties are only present when a backtrace capture is required, which is why it was chosen for implementation. Recall that opcodes in the interpreter consist of native functions that call each other: To unwind the stack it suffices to iteratively return from opcodes, capturing data about each activation in `op_Call` and `op_Entry` instructions, which is then used during recompilation and on-stack-replacement. Capturing the stack by returning form functions also makes on-stack replacement mandatory for resuming execution with the correct machine state. A diagram of the unwinding process is displayed in Figure 2.

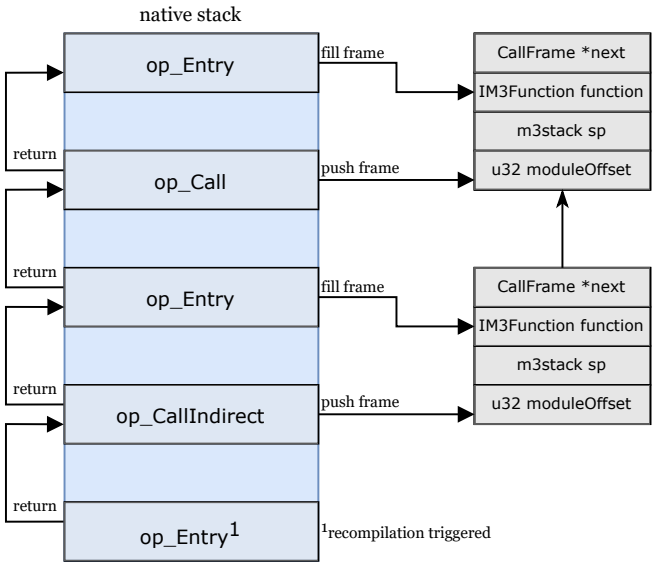


Figure 2: Backtrace capture during stack unwinding

4.2 OPTIMIZATION & INTEGRATION

After the entry of a hot function triggered recompilation and the call stack was unwound up to a suitable recompilation target, functions may be reoptimized with consideration of data captured by the profiling system.

Execution of the interrupted activations should then be continued using the newly compiled code to avoid executing, for instance, the unoptimized functions in a hot loop. The mechanism that facilitates swapping active invocations is called on-stack replacement.

4.2.1 *On-Stack Replacement*

Functions on a call stack can only be replaced if a mapping between the interrupted continuation of each function and an equivalent ex-

ecution state for the recompiled versions exists. The specific data required for such mapping varies depending on bytecode semantics and implementation details, but the following approach has been extensively cited since OSR's description for the debugging of SELF [24]: The location of local variables can be encapsulated in a "Scope Descriptor". A Scope Descriptor is generated for basic blocks during compilation and contains a mapping of specific local data locations to abstract bytecode locations and specify the surrounding scope. In addition, execution can only be interrupted at "Safepoints" or "Interrupt points"; Safepoints imply a mapping between intermediate computations such as register values to the equivalent positions on the abstract value stack. Because there may not be any valid mapping between the executing code and the virtual machine in between safepoints, functions can not be replaced in these sections.

Generating and storing metadata to allow for on-stack replacement is said to entail considerable memory overhead [1, 12]. With little memory being available on most microcontrollers, the amount of generated metadata has to be minimized. Next to cautious implementation, reducing the number of safepoints inherently decreases the amount of required bookkeeping.

Wasm3's extension can only trigger re-optimization at 'call' and 'call indirect' instructions. For either instruction, a bidirectional mapping between the physical program counter and a WASM module's bytecode offset exists. Because Wasm3's `op_Entry` instruction always allocates its own stack frame and transfers execution to the function's body, it must too be considered a safepoint. However, because the instruction's location is implicitly known, no additional metadata needs to be generated. Other data, such as the location of stack values that would be contained in a scope descriptor, is currently not required, because Wasm3 only supports recompiling functions to equivalent intermediate code that uses the same slot locations. However, once native code generation, inlining, or other code transforming optimizations were to be added, more metadata will have to be recorded to correctly map temporary and named values.

The bidirectional mapping between compiled and abstract code is implemented as two openly addressed hashtables stored per runtime. During compilation, a new entry is added to both tables whenever a 'call' or 'call indirect' opcode is emitted; the code implementing the insertion of new entries is shown in Listing 4.

To replace call frames on the stack, the bytecode offset is retrieved for the program counter during stack unwinding. Once replacement functions have been compiled, the `wasm2Pc` table is accessed to retrieve the now updated program counter and continue execution at the safepoint. In their original work, Hölze et al. describe an on-stack replacement procedure that creates new stack frames for every activation and fills in fields such as the return address and local variables. Wasm3 employs a more flexible strategy that does not require architecture-dependent stack modification: A stub function traverses the backtrace and emulates each caller's behavior, continuing execution of the code path following the interrupted safepoint as if the entire stackframe

```
typedef struct M3Runtime
{
    //...
    m3_table_u32_M3OSRPoint  wasm2Pc;
    m3_table_pc_t_u32        pc2Wasm;
    //...
} M3Runtime;

void  EmitMappingEntry  (IM3Runtime i_runtime, IM3CodePage i_page, u32
↪  i_moduleOffset, u8 len)
{
    pc_t pc = GetPagePC (i_page);
    m3_table_pc_t_u32_set(&i_runtime->pc2Wasm, pc, i_moduleOffset);
    m3_table_u32_M3OSRPoint_set(&i_runtime->wasm2Pc, i_moduleOffset,
    ↪  (M3OSRPoint) {pc, pc + len});
}
```

Listing 4: Wasm3 runtime struct including two hashtables for OSR

had been constructed. This is enabled thanks to the directly threaded code; with every instruction being a C procedure, it is easy to jump into the middle of a code page without needing to modify the VM's state. This method could also be used for replacing native code on-stack if the stub were to update the required stack- and return address registers before jumping into the code. To achieve OSR, the stub procedure jumps to the correct resumption point of every function on the stack in the correct order; evaluating and transferring return values between stackframes.

Figure 3 outlines the process of on-stack replacement in Wasm3, notice that the 'call' instruction of the first frame is executed because `$func3` triggered recompilation on its entry, but for any subsequent stack frames execution continues on the following opcode. To allow for recursive on-stack replacement, the stub takes ownership of the current backtrace so that recompilation may take place inside of an activation that itself has been invoked by the OSR stub in a nested manner.

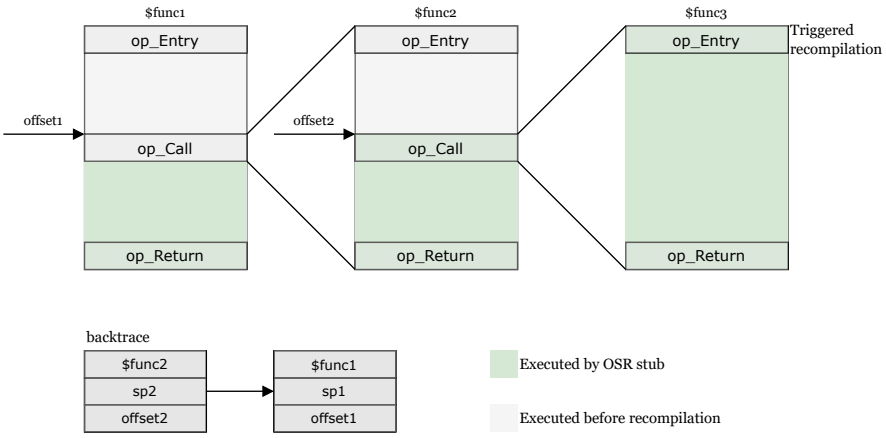


Figure 3: On Stack Replacement

4.2.2 *Recompilation*

The recompilation of WebAssembly functions occurs before their on-stack replacement. Unlike OSR, during recompilation, the captured backtrace is traversed from callers to callees, since it allows for potentially inlined code to be compiled with the surrounding context. When Wasm3 recompiles a function, handlers responsible for compiling each WebAssembly bytecode instruction can be swapped out to allow for different behavior. For instance, on function compilation, the `op_Entry` instruction is replaced with a variant that disables the invocation frequency calculation. Recompilation of a function resets any profiling data associated with it. To avoid allocating redundant memory on each compilation pass, internal data structures were adjusted to allow for the freeing of code pages associated with WASM functions.

Because 'call' instructions take a pointer to the target's `op_Entry` instruction as an immediate argument, all call sites targeting a recompiled function have to be updated after recompilation completes. A linked list was introduced to every function descriptor that stores the Wasm3 code page pointer of any related call sites, which is populated during each compilation pass. We need to store pointers to specific instances of the WebAssembly 'call' operation due to the possibility of having multiple, recompiled versions of a function active on the stack. If we instead used abstract WASM bytecode offsets to locate the most recently compiled code with our safepoint map, old but still active instances could not be patched to refer to the new version and may try to call freed memory.

At its current state, Wasm3 does not include a full native code generation backend; functions are transformed to the same intermediate format during recompilation. It is expected that the memory requirements will be higher with native code generation due to the compiler's overhead.

4.2.3 *Polymorphic Inline Caches*

Originally developed as a dynamic optimization [13], and later extended to gather profiling data [23], Polymorphic Inline Caches can be used at dynamic dispatch sites to record the times a specific target was called and to reduce dispatch overhead.

A WebAssembly interpreter has to run a function lookup routine when an indirect call is executed, indexing into a function table with the top stack operand. The overhead associated with function lookup can be reduced by introducing inline caches at the call site: The call to the lookup routine is replaced with a direct jump to the previous target's code pointer. To preserve correctness, a jump to a dynamic target has to be guarded by comparing the top stack value. Should the table index differ from the currently cached function, the lookup function will be used. An issue with such inline caching is that, while

reducing dispatch time, it cannot be used to count the number of invocations for any call target but the most recent.

Polymorphic inline caches extend the above concept by storing cached pointers out of line in dynamically allocated memory. Each cached target is ordered by its sequence of occurrence; should a guard detect a wrong table index, its value is compared with the next entry. If none of the cached functions fit the current index, a `backPatch` routine is called that performs the following steps:

1. The polymorphic inline cache is reallocated with additional space and its metadata is updated.
2. All relative pointers of the inline cache are recalculated and patched.
3. Function lookup is performed and a new entry for the target is added to the cache.
4. The pointer to the polymorphic inline cache at the call site is updated with its new location.

Using polymorphic inline caches, dynamic call sites can be associated with the invocation counts of targets by storing and updating a counter in each cache entry.

To increase execution speed, Wasm3's inline caches are generated as native machine code and its metadata, table indices and function pointers are stored inline using immediate instructions. The interpreter, instead of performing function lookup, jumps into the polymorphic inline cache. One complication of dynamically generating native code on resource-constrained devices stems from the Harvard architecture that many microcontrollers, including the ESP-32C3, adhere to. Because data- and instruction buses are separated and mapped to different regions of linear memory, addresses have to be translated between both regions when generating instructions that reference data or code. Furthermore, many instruction architectures including Armv7 have incoherent data- and instruction caches: Every dynamically generated instruction has to be flushed from the data cache and invalidated in the instruction cache. The native polymorphic inline cache also has to pass on or load any arguments that are required by the invoked WASM function.

Figure 4 illustrates the abstract layout of polymorphic inline caches in Wasm3 and how they affect native control flow. In subfigure 1, the 'call indirect' operation received the table index '3' on the WASM stack, which has not been used as an operand before. The processor jumps to the first cache entry and falls through to the `backPatch` routine. A following indirect call in subfigure 2, again using table index '3', falls through the first cache entry and executes the body of the entry added by the `backPatch` function.

At its current state, Wasm3 does not support the WebAssembly proposal for reference types[1] , which adds the ability to change function

1 https://github.com/WebAssembly/reference-types/blob/master/proposals/reference-types/Overview.md

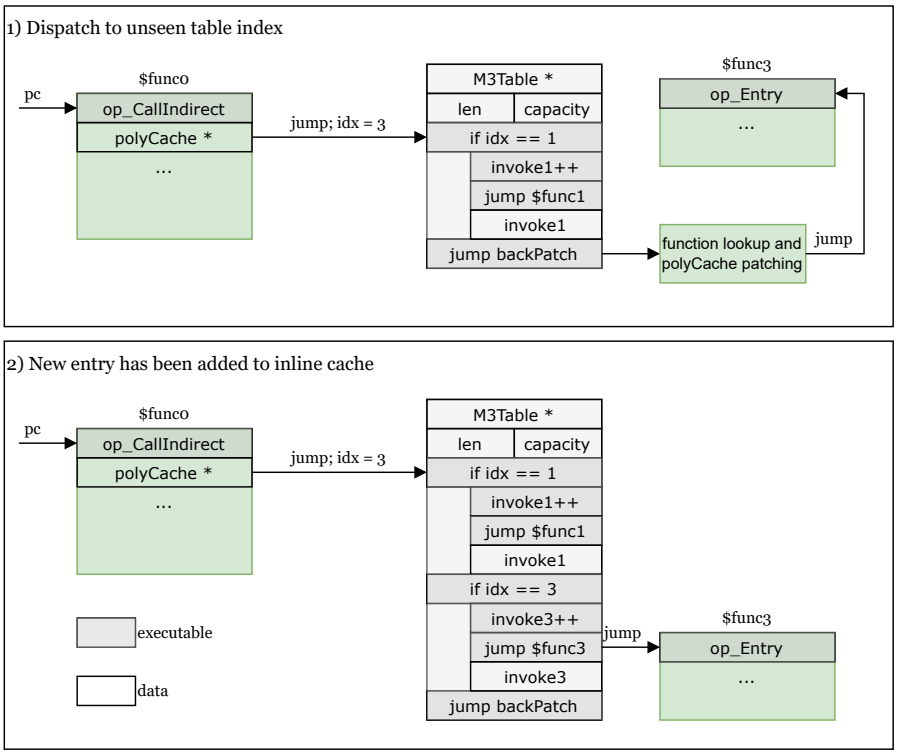


Figure 4: Layout and control flow using a polymorphic inline cache

table entries at runtime. Should support be added, the implementation of PICs would have to be adjusted to retain the correctness of a program; either of the following two changes would suffice:

- One can introduce an additional indirection between a cache entry and the table entry.
- Keeping track of every PIC that has cached a table index, when a table is modified, all cache entries are patched with the new content.

Wasm3's current PIC implementation already supports patching cached entries, because it is required for updating pointers during recompilation. When the `backPatch` procedure finds the target function, it inserts a list entry to the callee's metadata, similar to the compilation of direct calls described in Section 4.2.2.

4.3 GARBAGE COLLECTION

Implementing the above techniques necessitates the use of various dynamic, runtime data structures. In particular, each generation of compiled code and its associated metadata take up considerable memory. Table 2 contains the important types of data dynamically generated by Wasm3's dynamic optimization system. To run complex, continuous, dynamically optimized programs on our low-memory target environments, garbage collection should itself be memory efficient and not rely on context-switching processor capabilities.

Under these constraints, Wasm3's garbage collection algorithm attempts to free any allocated memory as soon as permissible for an object's lifetime. Garbage collectors can be measured using various criteria, such as memory and processing overhead, throughput, or pause times. Garbage collection is a vast topic of its own, with numerous criteria by which algorithms can be judged, e.g. complexity, pause latencies, or throughput and is beyond the scope of this evaluation. Wasm3's current garbage collection method only possesses minimal functionality and was not fine-tuned for particular metrics. Wilson gives a comprehensive overview of popular garbage collection techniques in his survey [54].

SYSTEM	DATA	LIFETIME
Reoptimization	Compiled Function Code	Until a newer version exists and the last activation finishes
Polymorphic Inline Caches	Code Stubs	While the encompassing code lives
Reoptimization	Per-Function call site list	While the call site code lives
On-Stack-Replacement	Safepoints-to-Wasm hash tables	Entire program execution time (per table); Function lifetime (per entry);
Profiling	In-Line Metadata (e.g. call frequencies)	While the associated function or PIC lives
On-Stack-Replacement	Backtrace	While any of the functions replaced on-stack are still active

Table 2: Dynamic data structures used by Wasm3 extension

The dynamic data associated with a function's compilation cannot be freed once the function has been recompiled because at the time of recompilation, an invocation of the old code may exist on the call stack. We need to track whether a version of code is currently active or not, at which point we can free associated data. To store this information, we check if an invocation of a function is the first to be placed on the call stack in the `op_Entry` instruction. If so, we allocate space on the native stack and move pointers to dynamic function data out of the **`struct`** `M3Function` onto the stack.

To account for separate recompiled versions of a function being nested on the call stack, we additionally associate a monotonic recompilation number with each function. Exiting a function uses the number to check if it has been recompiled during its runtime and releases the data previously stored on the stack accordingly.

With this manual garbage collection approach, memory is made available as soon as possible. The collection routine also removes any table entries associated with safepoints and deletes list entries in the function's callee's metadata. The code responsible for collecting garbage when entering and exiting a function is outlined in Listing 5.

Some features, such as polymorphic inline caches, need to access the dynamic data pointers saved on the stack during execution to record new call sites. To make the stack-local data available further down the call chain of the `op_Entry` instruction, a pointer to it is stored in the function's descriptor `struct M3Function`. The pointer allows for making runtime modifications to dynamic metadata while it is captured in a function's root invocation.

```
M3CodeOwnership ownedPages = TakeOwnership(function);
if (!function->activeCodePage || ownedPages.pagesOpen) {
    function->activeCodePage = &ownedPages;
}
m3ret_t r = nextOpImpl();
if (function->recompilation > ownedPages.recompilation) {
    ReleaseGarbage(function, &ownedPages);
} else {
    ReleaseOwnership(function, ownedPages);
}
function->activeCodePage = ownedPages.lastActive;
```

Listing 5: Garbage collection code addition to op_Entry

5 PROFILING

5.0.1 *Methodology*

The developed dynamic optimization framework was profiled on the ESP32C3 and the MIMXRT1170-EVK. The methods and WebAssembly modules were chosen to record data on two separate attributes: memory characteristics and processing overhead. Both features are useful for an initial evaluation of employing a complex, adaptive WASM optimization framework on SoCs.

The profiling system of Wasm3 was parameterized to trigger re-optimization at 15000 invocations, with a counter half-time of four seconds.

5.0.1.1 *Processing Overhead*

The CoreMark[1] benchmark is designed to assess a microcontroller's speed using a single, comparable number. CoreMark was compiled to WebAssembly bytecode and executed using the modified Wasm3 interpreter with required bindings, such as timing functions, provided as module imports. CoreMark is a suite of benchmarks containing the following algorithms: list processing, common matrix operations, state machine operations, and cyclic redundancy checking. A total of 25 executions per benchmarked configuration were taken to account for statistical errors. Every execution included a complete reinitialization of the interpreter.

5.0.1.2 *Memory profiling*

Within the context of dynamic compilation, the runtime behavior of memory allocations and deallocations during recompilations is particularly interesting. To not only record the overall heap usage, but the frequency, size, and time distribution of the interpreter's allocations, a heap trace was recorded for programs of varying complexity on both hardware platforms. Because of a lack of WebAssembly-compatible benchmarks that have sufficient code size but do not require platform-specific bindings or an operating system, two benchmarks were developed for the purpose of collecting heap data. Large module sizes are advantageous because they allow measuring the memory impact of having frequent recompilations.

Each data point was transmitted to the development host using a UART connection at the moment of occurrence. As a result of UART's bandwidth and latency, the program took considerably longer to finish execution under active heap tracing. The order of allocations to

1 https://www.eembc.org/coremark/

each other is not affected by this, but the absolute timestamps of all memory operations are skewed non-linearly by this overhead.

Every feature discussed in Chapter 4 was included when profiling the heap with dynamic optimization enabled.

5.0.1.3 *WebAssembly Programs*

BENCHMARK	INSTRUCTIONS	FUNCTIONS	INDIRECT CALLS
Serde	2718	24	0
CoreMark	3771	15	0
Interp	238633	2427	222

Table 3: Benchmark module characteristics

Table 3 outlines the different programs used when taking measurements. The 'CoreMark' benchmark was preexisting, but 'Serde' and 'Interp' were developed for profiling Wasm3 on our target hardware as described in Section 5.0.1.2. To evaluate the influence of the dynamic optimization framework, we require WASM modules with multiple functions and indirect calls. The Serde and Interp benchmarks perform the following functions:

SERDE iteratively serializes and deserializes structures to and from memory, comparing the result to the original data.

INTERP instantiates and executes the 'Rhai' interpreter. The Rhai interpreter then executes a script that calculates the number of primes smaller than N. This benchmark, by including an interpreter, allows for testing the execution of complex constructs such as dynamic dispatch.

5.0.2 *Processor Profiling*

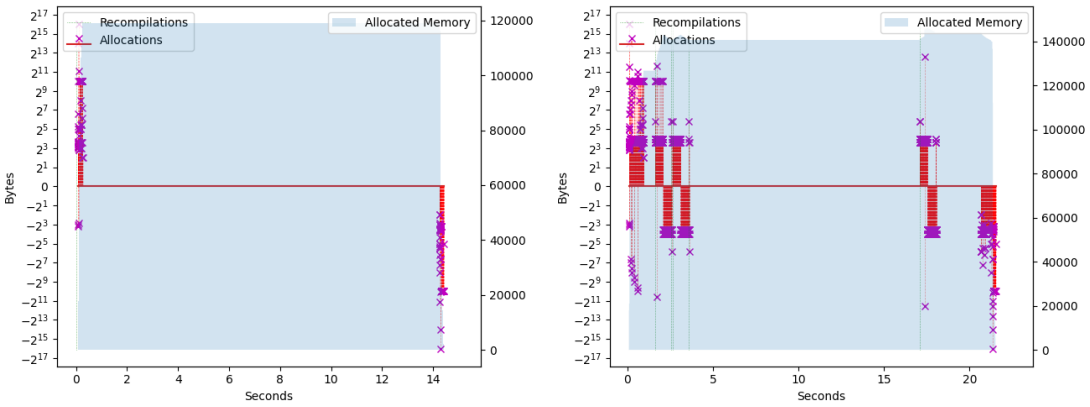
The average CoreMark result for each benchmarked feature-set is shown in Table 4. The only noticeable performance decrease coincides with the use of profiling and OSR, which are active in every configuration apart from 'All disabled'. The runtime overhead of enabling recompilations correlates with the complexity of the compilation and code generation; with the single-pass compiler of the Wasm3 interpreter, recompilations did not have a considerable effect on CoreMark results.

5.0.3 *Memory Profiling*

In Figure 5 and Figure 6 every allocation, deallocation, and reallocation is represented as a stem. The cumulative amount of memory allocated by the runtime is displayed in blue and measured using the right vertical axis. Note that the execution time of traced programs

CONFIGURATION	I.MXRT1170	ESP32C3
All enabled	88	20.16
No polycaches	87.85	20.18
No recompliation	88.82	20.49
No polycaches & no recompilation	88.66	20.44
All disabled	89.83	22.95

Table 4: CoreMark results



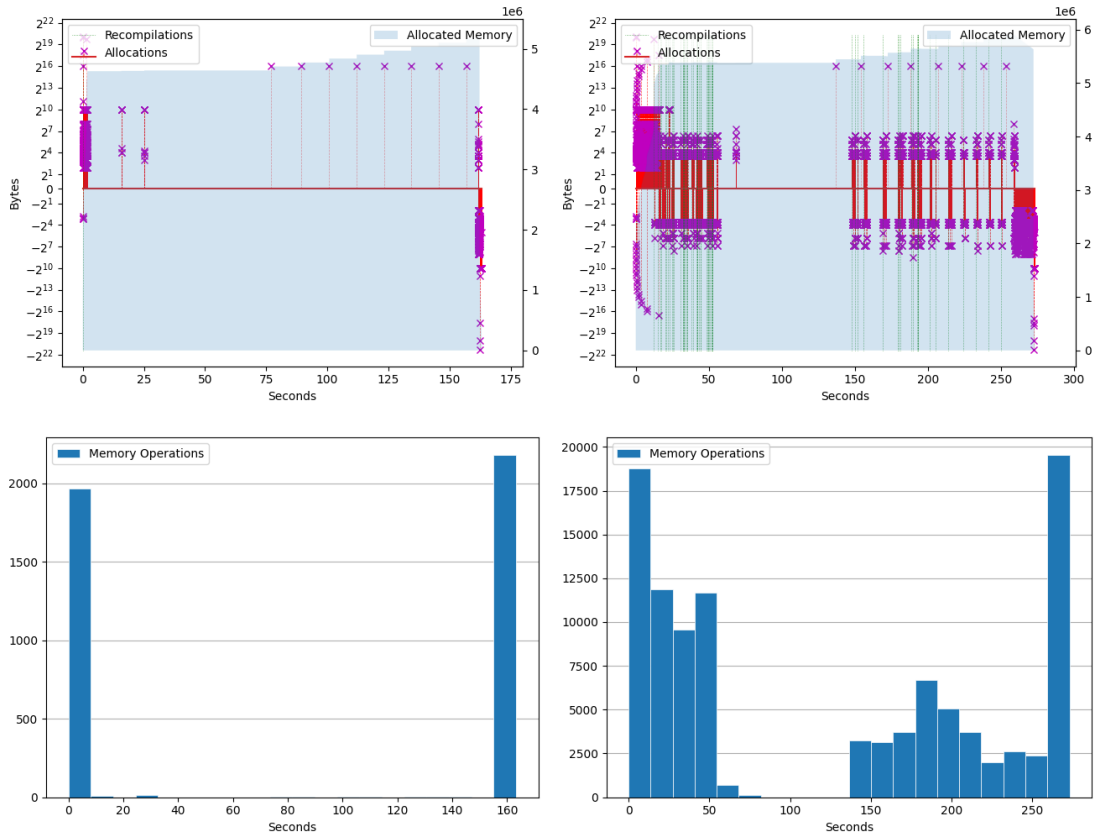
(a) No dynamic optimization

(b) Dynamic optimization

Figure 5: Memory Trace: ESP32C3 'Serde'

is slower when more memory operations are performed because of transmission delays. Comparing the differences between the profiles with optimizations enabled and disabled shows some interesting features:

- Much of the increase in total memory consumption is related to the additional metadata ancillary to the optimization framework. This is seen when comparing the line graph after the initial compilation pass but before any recompilations have taken place, at which point memory usage is higher with enabled optimization. Notably, the recompilations throughout a program's runtime do not considerably increase memory use, they free and allocate similar amounts of memory.
- Although the total memory consumption is only fractionally higher, dynamic optimization is associated with a considerable amount of additional memory operations. Execution of the 'Interp' benchmark under dynamic optimization performed 104811 memory operations, as opposed to 4176. As visible by the histogram, memory operations are distributed throughout the program's execution with optimization enabled, while being limited to runtime initialization and destruction without them.



(a) No dynamic optimization

(b) Dynamic optimization

Figure 6: Memory Trace: i.MXRT1170 'Interp'

- As seen with previous dynamic compilation systems, recompilations are more frequent during the startup of a program [45]. Since the program optimizes functions primarily at the start of execution, delays and pause times may be higher throughout that timeframe.

The memory (de-)allocations performed when executing the 'Interp' benchmark are classified in Table 5

CONFIGURATION	MEM-OPS	MEM-OPS EXCL. INITIALIZATION	MEM-OPS PER RECOMPILATION
All enabled	104811	67184	1101
All disabled	4176	NA	NA

Table 5: 'Interp' memory statistics

6 DISCUSSION

6.1 INTERPRETATION

The results obtained in Chapter 5 show that an interpreter framework required to enable dynamic- and speculative optimization of WebAssembly is feasible on resource-constrained MCUs and SoCs. The overall memory consumption of the interpreter did not considerably increase from allocating ancillary structures for metadata and recompiling functions. Total memory consumption can be influenced by changing the recompilation strategy of Wasm3: because a mapping between abstract WebAssembly bytecode and compiled code is required for all function safepoints, extending the potential recompilation sites to additionally include other instructions would increase memory use. Conversely, if we disable OSR and instead waited for all invocations to exit, we would not require memory for storing safepoint data at each `op_Call` instruction. For a basic WebAssembly dynamic optimization framework, supporting de-optimization and OSR at call sites suffices. A likely reason for supporting safepoints beyond the current implementation would be to allow for guarded speculative optimizations, as every guard needs to function as a safepoint.

The disparity between the speed of 'All disabled' and other configurations show that the profiling system of Wasm3, namely recording invocation and call edge frequencies, induce the most noticeable performance penalty. Choosing to sample a WebAssembly module instead of eagerly profiling every function invocation would likely decrease the associated costs. In a finalized dynamic optimization system, including native code generation, profiling for optimized functions should be enabled on demand based on other observances, as implemented by previous runtimes described in Section 2.3.1.5.

As visible in Figure 6 and Table 5, unlike the total memory consumption, the number of allocations and deallocations performed by the interpreter increased considerably. This number can be lowered by choosing different data structures for storing metadata and recompiled functions: The Wasm3 extensions, except for the mapping of safepoints described in Section 4.2.1, store additional metadata in linked lists that allocate each node separately. We could replace most lists with hash tables or dynamic arrays, decreasing the number of allocations performed at the cost of potentially more wasted memory. Whether such a tradeoff can increase resulting performance depends on different factors such as the hardware platform, the type of memory used, and the allocation strategy.

To conclude, although the experimental extension of Wasm3 developed in this paper allows for dynamically optimizing WASM programs on the target environments, known engineering practices can further improve and tailor the runtime to increase performance.

6.2 CODE GENERATION

Of course, on top of the features added to Wasm3, a complete Just-In-Time compilation system typically generates native machine code. Because WebAssembly functions are only recompiled using the pre-existing Wasm3 compiler, which generates threaded code in a single pass, the cost of recompilation is lower than if we were to generate machine instructions. However, while not being able to directly measure the impact of native code compilation, we can refer to previous work to evaluate the potential and overhead it would entail.

The following sections discuss the expected cost and benefit of extending Wasm3 to a full JiT-supporting interpreter.

6.2.1 *Cost of code generation*

Depending on the chosen intermediate representations, optimizations, and algorithms for emitting instructions, the speed and space requirements of code generation can widely vary. As a general rule, the more optimized constructed code has to be, the higher the associated costs are.

Prominent JiT compilers, such as google chrome's V8 [6] use costly techniques to generate their most optimized compilation tiers. Nießen et al. found that TurboFan, V8's optimizing compiler, takes an average of 500 to 1000 ms to compile a WebAssembly module of 1 MB; the compilation time scaling linearly with module size [33]. Their analysis of memory consumption shows that compilation uses between 32 and 64 MB for modules up to around 256 KB. However, memory consumption scales faster than the corresponding size of the module. Finally, TurboFan produces machine code that is usually two to three times larger than the source's WASM bytecode length.

Nießen et al's analysis was conducted using a workstation equipped with a powerful processor and large amounts of DRAM; code generation has to be far more efficient in an MCU-compatible JiT compiler. The blowup factor of resulting native code, as compared to source bytecodes, is less of a concern. Currently, Wasm3 translates most bytecodes to a function pointer that is 4 bytes wide on targetted platforms. As such, the IR of Wasm3 already has a comparable code size increase, which the memory profile shows to work with little available memory.

In previous work, fast, efficient code generation in JiT compilers was facilitated by the use of fewer, less costly steps in the compilation pipeline. For instance, linear scan register allocation requires only two passes over the IR to allocate registers. MicroJiT, an efficient Java bytecode compiler by Chen and Olukotun [10], produced native code that, in some cases, could rival the performance of the Sun server compiler at the time, while requiring substantially less time and memory. Their compiler required an average of five thousand instructions per processed bytecode, faster than Sun's server compiler by a factor of 12. The implementation also required little dynamic memory, allowing

for the compilation of Java methods up to 1 KB with a memory buffer of 250 KB.

While a compiler such as MicroJiT can produce optimized code without much overhead, its requirements might still be incompatible with the specifications of low-cost MCUs like the ESP32C3. For these devices, dynamic memory use and compilation pauses can be further reduced by employing baseline compilers that perform little to no optimizations when generating instructions. For example, the Sun client compiler, integrated into the Java runtime as a baseline compiler, used only half as much dynamic memory as MicroJiT.

To summarize, while the most aggressive compilers like TurboFan are likely to be incompatible with small SoCs, one can implement native compilers that produce somewhat less optimal code with far fewer resource constraints. Further tradeoffs between code performance and compilation efficiency can be made depending on the specific target device and its specifications.

6.2.2 *Benefit of code generation*

Extending Wasm3's dynamic optimization framework with native code generation promises considerable performance benefits, even if the native compiler cannot apply all state-of-the-art transformations and algorithms.

Titzer compared the execution performance of different WebAssembly runtimes on benchmarks of varying lengths to observe the tradeoff between fast and optimal code generation [47]. He found that, with increasing program runtimes, Just-in-Time compilers gain an advantage over Wasm3, which was the fastest WASM interpreter he tested. Interestingly, even baseline compilers such as V8's Liftoff are still competitive when compared to high-tier compilers; unlike Wasm3, which took up to 10 times as long as TurboFan, and 5 times as long as Liftoff to complete long-running benchmarks. Chen and Olukotun observed similar results when comparing the speedup of different Java JiT configurations [10].

As such, we can expect a substantial performance improvement with the addition of code generation, regardless of the tradeoff made between optimization and resource usage as discussed in Section 6.2.1. The time saved on bytecode execution will likely compensate for the overhead from profiling and other systems discussed in Section 6.1.

6.3 FUTURE WORK

The results obtained from implementing an MCU-compatible, WebAssembly dynamic optimization framework show that such use is viable and with native code generation would likely outperform an interpreter in general usage. In this section, potential further work on the topic and new questions that arise from Wasm3's evaluation are discussed.

6.3.1 *Next Steps*

An obvious improvement to the experimental Wasm3 extension developed in this thesis is native code compilation. Comparing code generation's performance impact in Section 6.2 implies that fully implementing a WebAssembly JiT for constrained platforms would yield substantial runtime performance gains. In addition, working code generation is the foundation for further research on the topic.

With the complexities involved in profiling and native compilation, I expect it to be beneficial to rework Wasm3's intermediate representation or to develop a new WebAssembly research VM. Wasm3 was developed as an 'Interpreter-Only' runtime and does not have a compilation pipeline that supports storing detailed profiling data about instructions or to analyze the program's data- and control flow. Instead, the single-pass compiler of Wasm3 directly converts bytecode to executable code pages. While this model was sufficient for the purpose of this thesis, adding code generation and speculative optimizations would likely benefit from a different bytecode representation.

6.3.2 *Open Questions*

Extending Wasm3's capabilities, or developing a new VM with complete JiT-compilation support, allows us to answer further questions about dynamic- and speculative optimization of WebAssembly on SoCs:

- How much do different WASM source languages benefit from JiT compilation and speculative optimization?
- Can we place an upper bound on compilation latencies?
- What are the security implications of dynamically compiling WebAssembly to machine code on embedded devices?
- Would separate tiers of optimization, as present in advanced virtual machines, be beneficial?

Regarding the first point, most existing research evaluates the impact of JiT compilation for runtimes that support either a single, or a small set of similarly structured, source languages. The Java virtual machine is an example of this: Most languages targetting the JVM make use of concepts such as dynamic dispatch. With the heterogeneous landscape of WebAssembly-compatible languages, ranging from C to JavaScript, languages may benefit differently from JiT compilation.

Placing a bound on dynamic WebAssembly compilation latencies would be especially relevant for real-time applications. Perhaps an offline analysis of WASM bytecode could be used to attach a recompilation cost to every function, and the interpreter can evaluate these to make recompilation decisions.

The third point relates to the strong memory guarantees WebAssembly makes about a module: Accesses are always restricted to predeclared value locations or linear memory sections with known size. Executing native, just-in-time compiled code should preserve as many guarantees about a program's dynamics as possible. One of WebAssembly's advantages over native execution is the safety and memory guarantees achieved through its module sandboxing. If just-in-time compiled code couldn't easily enforce the same restrictions, any performance increases may be nullified.

6.4 RELATED WORK

WebAssembly is a relatively new development, and most research on it and its runtimes has been published over the past few years.

Wang, in his 2022 study of standalone WebAssembly runtimes, evaluates the characteristics and performance of five separate WASM runtimes, some of which perform JiT compilation [51]. Notably, he compares each runtime to native code execution and the impact of JiT compilation.

Nießen et al. develop a method of code caching for established WebAssembly runtimes to improve performance and reduce resource usage [33]. Their paper also includes a fine-grained evaluation of V8's compilation profile regarding memory and processing time costs.

Titzer presents a fast in-place WASM interpreter, that improves upon the startup delay of many existing interpreters and baseline compilers [47]. The article also contains a comprehensive comparison of existing WebAssembly interpreters and JiT compilers, including the V8 and Spidermonkey execution engines.

Xu and Kjolstad describe their 'copy-and-patch' compilation technique and showcase its application on WebAssembly bytecode [55]. The strategy aims to reduce the cost of compilation while achieving high-quality compilation results. For this, they leverage existing compilers to generate templates, which are instantiated at runtime with concrete values and emitted.

6.5 CONCLUSION

The extensions added to Wasm3 follow known dynamic optimization strategies mainly developed through research on the Smalltalk, SELF, and Java languages. The extensions were developed with resource-constrained target hardware in mind. Some data structures and algorithms, like the invocation counter decaying method or the on-stack replacement technique, were adjusted to be viable in resource-constrained environments, while some strategies, like using task-based sampling methods, were excluded from Wasm3 due to likely platform complications.

Evaluation of memory and processing performance shows that the underlying system to enable dynamic optimization of WebAssembly are feasible on MCUs. As discussed in Section 6.2, finalizing the JiT

by adding native code compilation is expected to allow for significant increases to WASM's performance, although the implemented code generation has to be carefully designed to stay within resource bounds.

Continuing past this initial evaluation, there are further questions concerning efficient JiT compilation of WebAssembly left to answer in future research.

BIBLIOGRAPHY

[1] Ole Agesen and David Detlefs. *Mixed-Mode Bytecode Execution*. Tech. rep. USA, 2000.

[2] ARM Limited. *Cortex-M4 Technical Reference Manual*. Version ropo. Dec. 22, 2009.

[3] ARM Limited. *Armv7-M Architecture Reference Manual*. Version E.e. Feb. 25, 2021.

[4] M. Arnold, S.J. Fink, D. Grove, M. Hind, and P.F. Sweeney. "A Survey of Adaptive Optimization in Virtual Machines." In: *Proceedings of the IEEE* 93.2 (Feb. 2005), pp. 449–466. DOI: 10.1109/jproc.2004.840305.

[5] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. "Adaptive optimization in the Jalapeño JVM." In: *ACM SIGPLAN Notices* 35.10 (Oct. 2000), pp. 47–65. DOI: 10.1145/354222.353175.

[6] Clemens Backes. *Liftoff: a new baseline compiler for WebAssembly in V8*. Ed. by Clemens Backes. Aug. 20, 2018. URL: https://web.archive.org/web/20220815095400/https://v8.dev/blog/liftoff.

[7] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. "Dynamo: a transparent dynamic optimization system." In: *ACM SIGPLAN Notices* 35.5 (May 2000), pp. 1–12. DOI: 10.1145/358438.349303.

[8] C. Chambers, D. Ungar, and E. Lee. "An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes." In: *ACM SIGPLAN Notices* 24.10 (Oct. 1989), pp. 49–70. DOI: 10.1145/74878.74884.

[9] Craig Chambers and David Ungar. "Making pure object-oriented languages practical." In: *ACM SIGPLAN Notices* 26.11 (Nov. 1991), pp. 1–15. DOI: 10.1145/118014.117955.

[10] Michael Chen and Kunle Olukotun. *Targeting Dynamic Compilation for Embedded Environments*. Tech. rep. July 2001. DOI: 10.21236/ada419605.

[11] Michał Cierniak, Guei-Yuan Lueh, and James M. Stichnoth. "Practicing JUDO." In: *ACM SIGPLAN Notices* 35.5 (May 2000), pp. 13–26. DOI: 10.1145/358438.349306.

[12] David Detlefs and Ole Agesen. "Inlining of Virtual Methods." In: *ECOOP' 99 — Object-Oriented Programming*. Springer Berlin Heidelberg, 1999, pp. 258–277. DOI: 10.1007/3-540-48743-3_12.

[13] L. Peter Deutsch and Allan M. Schiffman. "Efficient implementation of the smalltalk-80 system." In: *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '84*. ACM Press, 1984. DOI: 10.1145/800017.800542.

[14] Espressif Systems. *ESP-IDF Programming Guide*. Version 4.4.2. Aug. 2, 2022. URL: https://web.archive.org/web/20221231120936/https://docs.espressif.com/projects/esp-idf/en/v4.4.2/esp32c3/get-started/index.html (visited on 09/05/2022).

[15] Espressif Systems. *ESP32-C3 Technical Reference Manual*. Version 0.6. 2022. URL: https://web.archive.org/web/20221231120859/https://www.espressif.com/sites/default/files/documentation/esp32-c3_technical_reference_manual_en.pdf (visited on 09/04/2022).

[16] S.J. Fink and Feng Qian. "Design, implementation and evaluation of adaptive recompilation with on-stack replacement." In: *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* IEEE Comput. Soc, 2003. DOI: 10.1109/cgo.2003.1191549.

[17] Andreas Gal, Christian W. Probst, and Michael Franz. "HotpathVM: An Effective JIT Compiler for Resource-constrained Devices." In: *Proceedings of the 2nd international conference on Virtual execution environments - VEE '06*. ACM Press, 2006. DOI: 10.1145/1134760.1134780.

[18] Gabriel Gaspar, Peter Fabo, Michal Kuba, Juraj Dudak, and Eduard Nemlaha. "MicroPython as a Development Platform for IoT Applications." In: *Intelligent Algorithms in Software Engineering*. Springer International Publishing, 2020, pp. 388–394. DOI: 10.1007/978-3-030-51965-0_34.

[19] Damien George, Paul Sokolovsky, and contributors. *MicroPython documentation: Memory Management*. Ed. by Damien George, Paul Sokolovsky, and contributors. Aug. 18, 2022. URL: https://web.archive.org/web/20220818180929/https://docs.micropython.org/en/latest/develop/memorymgt.html.

[20] Dan Gohman. *WebAssembly System Interface*. Tech. rep. Bytecode Alliance, Nov. 12, 2019. URL: https://web.archive.org/web/20220824104852/https://github.com/bytecodealliance/wasmtime/blob/main/docs/WASI-overview.md (visited on 08/17/2022).

[21] Christian Häubl, Christian Wimmer, and Hanspeter Mössenböck. "Context-sensitive trace inlining for Java." In: *Computer Languages, Systems and Structures* 39.4 (Dec. 2013), pp. 123–141. DOI: 10.1016/j.cl.2013.04.002.

[22] Aaron Hilbig, Daniel Lehmann, and Michael Pradel. "An Empirical Study of Real-World WebAssembly Binaries." In: *Proceedings of the Web Conference 2021*. ACM, Apr. 2021. DOI: 10.1145/3442381.3450138.

[23] Urs Hölzle, Craig Chambers, and David Ungar. "Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches." In: *Proceedings of the European Conference on Object-Oriented Programming*. ECOOP '91. Berlin, Heidelberg: Springer-Verlag, 1991, pp. 21–38. ISBN: 3540542620.

[24] Urs Hölzle, Craig Chambers, and David Ungar. "Debugging optimized code with dynamic deoptimization." In: *ACM SIGPLAN Notices* 27.7 (July 1992), pp. 32–43. DOI: 10.1145/143103.143114.

[25] Urs Hölzle and David Ungar. "A third-generation SELF implementation." In: *Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications - OOPSLA '94*. ACM Press, 1994. DOI: 10.1145/191080.191116.

[26] Urs Hölzle and David Ungar. "Optimizing dynamically-dispatched calls with run-time type feedback." In: *ACM SIGPLAN Notices* 29.6 (June 1994), pp. 326–336. DOI: 10.1145/773473.178478.

[27] Martin Jacobsson and Jonas Willén. "Virtual Machine Execution for Wearables Based on WebAssembly." In: *13th EAI International Conference on Body Area Networks*. Springer International Publishing, 2020, pp. 381–389. DOI: 10.1007/978-3-030-29897-5_33.

[28] Kogge. "An Architectural Trail to Threaded-Code Systems." In: *Computer* 15.3 (Mar. 1982), pp. 22–32. DOI: 10.1109/mc.1982.1653970.

[29] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java® Virtual Machine Specification*. Feb. 28, 2013. URL: https://web.archive.org/web/20220709133447/https://docs.oracle.com/javase/specs/jvms/se7/html/index.html.

[30] Zoltán Majó. *Compilation in the HotSpot VM*. Ed. by Zoltán Majó. Oracle Corporation. 2015. URL: https://web.archive.org/web/20220719130212/https://ethz.ch/content/dam/ethz/special-interest/infk/inst-cs/lst-dam/documents/Education/Classes/Fall2015/210_Compiler_Design/Slides/hotspot.pdf.

[31] Niko Mäkitalo, Tommi Mikkonen, Cesare Pautasso, Victor Bankowski, Paulius Daubaris, Risto Mikkola, and Oleg Beletski. "WebAssembly Modules as Lightweight Containers for Liquid IoT Applications." In: *Lecture Notes in Computer Science*. Springer International Publishing, 2021, pp. 328–336. DOI: 10.1007/978-3-030-74296-6_25.

[32] Steven Massey and Volodymyr Shymanskyy. *Wasm3*. Aug. 31, 2022. URL: https://web.archive.org/web/20220827003837/https://github.com/wasm3/wasm3 (visited on 09/05/2022).

[33] Tobias Nießen, Michael Dawson, Panos Patros, and Kenneth B. Kent. "Insights into WebAssembly: Compilation Performance and Shared Code Caching in Node.Js." In: *Proceedings of the 30th Annual International Conference on Computer Science and Software Engineering*. CASCON '20. Toronto, Ontario, Canada: IBM Corp., 2020, pp. 163–172.

[34] NXP Semiconductors. *i.MX RT1170 MCU Family – Fact Sheet*. Jan. 8, 2021. URL: https://web.archive.org/web/20221005234514/http://www.nxp.com/docs/en/fact-sheet/i.MX-RT1170-FS.pdf (visited on 12/07/2022).

[35] NXP Semiconductors. *i.MX RT1170 Processor ReferenceManual*. May 2021.

[36] Oracle. *Java Card data sheet*. 2019. URL: https://web.archive.org/web/20220627130905/https://www.oracle.com/technetwork/java/javacard/overview/java-card-data-sheet-19-01-07-5250140.pdf (visited on 08/23/2022).

[37] Gregor Peach, Runyu Pan, Zhuoyi Wu, Gabriel Parmer, Christopher Haster, and Ludmila Cherkasova. "eWASM: Practical Software Fault Isolation for Reliable Embedded Devices." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.11 (Nov. 2020), pp. 3492–3505. DOI: 10.1109/tcad.2020.3012647.

[38] RISC-V Foundation. *Risc-V specification license*. Ed. by Risc-V contributing members. Feb. 2, 2017. URL: https://web.archive.org/web/20220904152132/https://github.com/riscv/riscv-isa-manual/blob/master/LICENSE.

[39] RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. Ed. by Andrew Waterman, Krste Asanovic, and SiFive Inc. Version 2019121. Dec. 2019.

[40] John R. Rose. "Bytecodes meet combinators." In: *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages - VMIL '09*. ACM Press, 2009. DOI: 10.1145/1711506.1711508.

[41] Andreas Rossberg. *GC Proposal for WebAssembly*. Tech. rep. W3C, Aug. 17, 2022. URL: https://web.archive.org/web/20220824104907/https://github.com/WebAssembly/gc/blob/main/README.md.

[42] Scott Logic. *The State of WebAssembly 2022*. Ed. by Colin Eberhardt. June 20, 2022. URL: https://web.archive.org/web/20220824105722/https://blog.scottlogic.com/2022/06/20/state-of-wasm-2022.html.

[43] Robbert Gurdeep Singh and Christophe Scholliers. "WARDuino: a dynamic WebAssembly virtual machine for programming microcontrollers." In: *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes - MPLR 2019*. ACM Press, 2019. DOI: 10.1145/3357390.3361029.

[44] Benedikt Spies and Markus Mock. "An Evaluation of WebAssembly in Non-Web Environments." In: *2021 XLVII Latin American Computing Conference (CLEI)*. IEEE, Oct. 2021. DOI: 10.1109/clei53233.2021.9640153.

[45] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. "Design and evaluation of dynamic optimizations for a Java just-in-time compiler." In: *ACM Transactions on Programming Languages and Systems* 27.4 (July 2005), pp. 732–785. DOI: 10.1145/1075382.1075386.

[46] Antero Taivalsaari and Tommi Mikkonen. "A Taxonomy of IoT Client Architectures." In: *IEEE Software* 35.3 (May 2018), pp. 83–88. DOI: 10.1109/ms.2018.2141019.

[47] Ben L. Titzer. "A fast in-place interpreter for WebAssembly." In: *Proceedings of the ACM on Programming Languages* 6.OOPSLA2 (Oct. 2022), pp. 646–672. DOI: 10.1145/3563311.

[48] Raoul-Gabriel Urma. "Alternative Languages for the JVM." In: *Java magazine* (July 2014), pp. 5–11. URL: https://archive.org/details/JavaMagazine2014.0708/mode/2up.

[49] Brian Vermeer. *JVM Ecosystem Report 2020*. Ed. by Brian Vermeer. Feb. 5, 2020. URL: https://web.archive.org/web/20220824104904/https://snyk.io/blog/jvm-ecosystem-report-2020/.

[50] W3C. *WebAssembly JavaScript Interface*. Ed. by Ms2ger (Igalia). Apr. 19, 2022. URL: https://web.archive.org/web/20220515191659/https://www.w3.org/TR/2022/WD-wasm-js-api-2-20220419/.

[51] Wenwen Wang. "How Far We've Come – A Characterization Study of Standalone WebAssembly Runtimes." In: *2022 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, Nov. 2022. DOI: 10.1109/iiswc55918.2022.00028.

[52] WebAssembly Community Group. *WebAssembly Specification*. Ed. by Andreas Rossberg. Aug. 11, 2022. URL: https://web.archive.org/web/20220824104926/https://webassembly.github.io/spec/core/.

[53] John Whaley. "A portable sampling-based profiler for Java virtual machines." In: *Proceedings of the ACM 2000 conference on Java Grande - JAVA '00*. ACM Press, 2000. DOI: 10.1145/337449.337483.

[54] Paul R. Wilson. "Uniprocessor Garbage Collection Techniques." In: *IWMM*. 1992.

[55] Haoran Xu and Fredrik Kjolstad. "Copy-and-patch compilation: a fast compilation algorithm for high-level languages and bytecode." In: *Proceedings of the ACM on Programming Languages* 5.OOPSLA (Oct. 2021), pp. 1–30. DOI: 10.1145/3485513.

[56] Sungjoo Yoo and A.A. Jerraya. "Introduction to hardware abstraction layers for SoC." In: *2003 Design, Automation and Test in Europe Conference and Exhibition*. IEEE Comput. Soc. DOI: 10.1109/date.2003.1253629.