Goto table of contents

DHBW
Duale Hochschule
Baden-Württemberg
Mannheim

THEMA STUDIENARBEIT

Exploring WebAssembly for versatile plugin systems through the example of a text editor

im Studiengang

TINF22IT1

an der *Duale Hochschule Baden-Württemberg Mannheim*

von

Name, Vorname: Hartung, Florian

Abgabedatum: 15.04.2025

Bearbeitungszeitraum:	15.10.2024 - 15.04.2024
Matrikelnummer, Kurs:	6622800, TINF22IT1
Wiss. Betreuer*in der Dualen Hochschule:	Gerhards, Holger, Prof. Dr.

Erklärung zur Eigenleistung

Ich versichere hiermit, dass ich meine Projektarbeit mit dem

THEMA
Exploring WebAssembly for versatile plugin systems through the example of a text editor

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Ort, Datum Unterschrift

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet, ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum defuturum, quas natura non depravata desiderat. Et quem ad me accedis, saluto: 'chaere,' inquam, 'Tite!' lictores, turma omnis chorusque: 'chaere, Tite!' hinc hostis mi Albucius, hinc inimicus. Sed iure Mucius. Ego autem mirari satis non queo unde hoc sit tam insolens domesticarum rerum fastidium. Non est omnino hic docendi locus; sed ita prorsus existimo, neque eum Torquatum, qui hoc primus cognomen invenerit, aut torquem illum hosti detraxisse, ut aliquam ex eo est consecutus? – Laudem et caritatem, quae sunt vitae.

Zusammenfassung

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet, ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum defuturum, quas natura non depravata desiderat. Et quem ad me accedis, saluto: 'chaere,' inquam, 'Tite!' lictores, turma omnis chorusque: 'chaere, Tite!' hinc hostis mi Albucius, hinc inimicus. Sed iure Mucius. Ego autem mirari satis non queo unde hoc sit tam insolens domesticarum rerum fastidium. Non est omnino hic docendi locus; sed ita prorsus existimo, neque eum Torquatum, qui hoc primus cognomen invenerit, aut torquem illum hosti detraxisse, ut aliquam ex eo est consecutus? – Laudem et caritatem, quae sunt vitae.

Contents

1 Introduction (4 pages)

1.1 Motivation & problem statement

1.2 Research question

Is WebAssembly the best technology choice for designing versatile plugin systems for text editors?

1.3 Methodology

2 Fundamentals (15 pages)

This section introduces theoretical and technical fundamentals used in this work.

2.1 Instruction set architectures

In the field of structured computer organization Tanenbaum defines a instrucion set architecture (ISA) as a level in a multilayered computer system [1]. The ISA level defines a machine language with a fixed set of instructions [1]. According to Tanenbaum the ISA level then acts as a common interface between the hardware and software. This allows software in the form of ISA instructions to manipulate the hardware [1]. Software written in a higher level machine language (Assembly, C, Java, …) can not be executed directly by the hardware. Instead higher level machine codes are compiled to ISA machine code or interpreted by a program, that is present in ISA machine code itself [1].

2.2 WebAssembly

2.2.1 Overview

WebAssembly (Wasm) is a stack-based ISA for a portable, efficient and safe code format. Originally it was designed by engineers from the four major vendors to enable high-performance code execution on the web [2]. However it is also becoming increasingly interesting for researchers and developers in non-web contexts. Some examples are avionics for Wasm's safe and deterministic execution [3], distributed computing for its portability and migratability [4] or embedded systems for its portability and safety [5].

What is special about Wasm is that it is a *virtual* ISA [6]. There is no agreed-upon definition for a virtual ISA, however the term *virtual* can be assumed to refer to an ISA that is running in a virtualized environment on a higher level in a multilevel computer[1]. We call this virtualized environment the **host environment** (used by the specification) or the **WebAssembly runtime** (used by most technical documentation).

TODO: Descripe Wasm in a multilevel computer system with a figure and give examples for levels

[1]There are a couple toy projects which have tried to execute Wasm directly. One example is the discontinued `wasmachine` project, which tried to execute Wasm on FPGAs.

2.2.1 Overview

WebAssembly code
WebAssembly runtime
Problem-oriented language level
Assembly language level
Operating system machine level
Instrucion set architecture level
Microarchitecture level
Digial logic level

Table 1: ***TODO: This figure needs annotations for compilation/interpretation and level numbers***. A multilevel computer system running Wasm code. Inspired by Figure 1-2 in [1]

WebAssembly code can be written by hand, just as one could write traditional ISA instructions (think of writing x86 machine code) by hand. Even though it is possible, it would not be efficient to write useful software systems in Wasm because the language is too simple. For example it provides only a handful of types[2].

WebAssembly also does not provide any way to specify memory layouts as it can be done in higher-level languages with structs, classes, records, etc. Instead it provides most basic features and instructions, which exist on almost all modern computer architectures like integer and floating point arithmetic, memory operations or simple control flow constructs. This is by design, as WebAssembly is mor of a compilation target for higher level languages[6]. Those higher level languages can then build upon Wasms basic types and instructions and implement their own abstractions like memory layouts or control flow constructs on top. This is analogous to the non-virtual ISA machine code in a conventional computer, which also acts as a compilation target for most low-level languages such as Assembly, C, or Rust. Most of these compiled languages like C, C++, Rust, Zig or even Haskell can be compiled to WebAssembly moderatly easily nowadays[3].

2.2.2 Execution model

This section presents the Wasm execution model and lifecycle of a Wasm module.

[2] Signed/unsigned integers, floating point numbers, a 128-bit vectorized number and references to functions/ host objects

[3] ***TODO: provide examples for compilers that can target Wasm***

2.2.2 Execution model

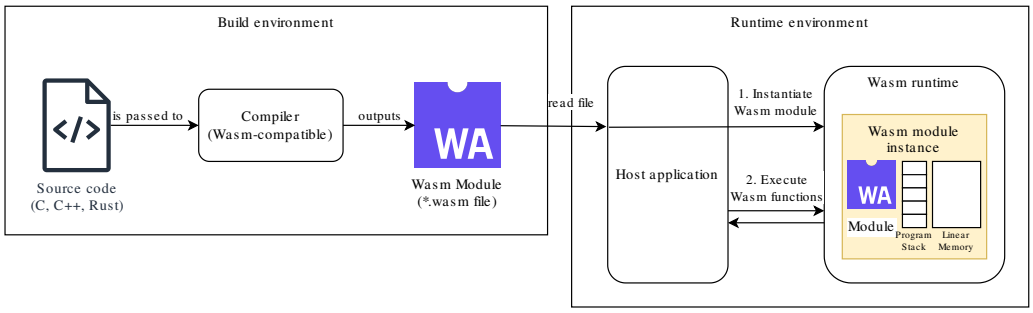

Figure 1: Flowchart for the creation and execution of a Wasm module from a higher-level language

Figure 1 shows the different stages a Wasm module goes through. The lifetime of a Wasm program starts with a developer writing source code. This source code can be written in an arbitrary programming language such as C, C++ or Rust. Then a compiler for that specific programming language creates a Wasm module. Compiling to Wasm requires explicit support from the chosen compiler.

- LLVM is used for a lot of projects
- What exactly is LLVM
- Example projects using LLVM

This Wasm module is fully self-contained in one singular `.wasm` file. It consists of different sections, that contain functions with their instructions, data segments, import- and export definition, etc.

The previously genertedWasm module can then be transferred to any target device or platform. In Figure 1 this device/platform is called the *runtime environment*. This runtime environment requires a Wasm runtime to be present. The Wasm runtime is able to parse the Wasm module file, instantiate a *Wasm instance* from it and provide an Application Programming Interface (API) for interaction with the Wasm instance. APIs can differ from one Wasm runtime to another. Some runtimes exist as standalone programs that can run Wasm modules comparable to how native binaries can be executes. Others are in the form of libraries, that can only be used from a host application to embed a Wasm runtime into them. These Wasm runtime libraries often provide common operations to the host application like calling Wasm functions, reading and writing operations for Wasm memories, linking mechanisms between Wasm modules, exposing host-defined functions for Wasm instances to call, etc.

2.2.2 Execution model

In a web context a server might provide this Wasm module to the client's browser, which contains a Wasm runtime[4] For distributed computing this Wasm module could be distributed among multiple different nodes regardless of their architectures[5]. Those nodes could then perform heavy computations and split work between each other by communicating through conventional methods like HTTP.

2.2.3 Design goals

Wasm was designed with certain design goals in mind. This section presents the most relevant design goals as necessary for this work. Each design goal is accompanied by related information from various papers and articles for a deeper understanding of each goal.

2.2.3.A Fast

Wasm is designed to be fast both during startup and execution [6].

Startup time is optimized through the Wasm bytecode format. Wasm's bytecode format is designed to make the runtime's work easier.

Why does Wasm provide fast startup time?

- Single-pass compiler, e.g. backwards jumps are disallowed (except for loops).

  Branches are referenced relativly by nesting depth per function.

This is done through a defined ordering of sections, e.g. by being parsable by a single-pass compiler

- Fast Runtime
  - Minimal set of opcodes
  - Parsable by a single-pass compiler: e.g. no backwards-jumps allowed (except loops), branches are referenced relativly by nesting depth per function (not absolutely with jumps to addresses)

TODO

An additional property of Wasm bytecode is that it can be either compiled, interpreted or just-in-time compiled by a runtime. This allows users to customize Wasm

[4]Most modern browsers come with a Wasm runtime: See https://caniuse.com/wasm for detailed information.

[5]This assumes that a Wasm runtime is available for those specific architectures. However here the system administrator could opt for different Wasm runtimes specifically tailored to each system. For example one might use an interpreter to avoid compilation complexity for compilers and JIT-compilation only on embedded devices.

runtimes for their specific needs and usecases. For example one could choose compilation to achieve faster execution at the cost of slow startup times due to compilation. On the other end users might prefer interpretation where execution speeds are less of a priority and fast startup times or relocatable runtime instances (see Section 2.2.3.G) are needed.

2.2.3.B Safe

- safe by default
- Sandboxed
  - Memory access is safe: Wasm is stack based & provides a linear memory, which can only be accessed through indices with bounds checks on every access.
  - Can only interact with host enviroment through functions that are explicitly exposed by the host
- Because of its simplicity: easier to implement a minimal working runtime than it is for higher level languages like C, C++, Java, Python, …
  - This again reduces risk of (safety-critical) bugs
- Properties like portability and safety are especially important in the context of the web, where untrusted software from a foreign host is executed on a client's device.
- This makes WASM, a sandboxed and fast execution enviroment, interesting for safety-critical fields like avionics (***TODO: ref?***, automotive (***TODO: ref*** https://oxidos.io/), ***TODO: what else?***
- Note: Applications inside Wasm can still corrupt their own memory.

TODO

2.2.3.C Portable

Wasm is designed to be able to be portable for a lot of different hardwares and platforms.

- minimal set of opcodes (172), that exist on all architectures(***TODO: insert screenshot of opcode table*** (+ proposals for more additional instructions like SIMD, atomics, etc.)

Independence of hardware:

- Desktop architectures
- Mobile device architectures
- Embedded system architectures

Independece of platform:

2.2.3.C Portable

TODO: opcode table

Figure 2: All WebAssembly opcodes. Opcodes for proposals are encoded by specific marker bytes, which indicate that the following opcodes are to be interpreted as different instructions.

- Browsers
- Other environments which only require some kind of a Wasm runtime

TODO

2.2.3.D Independence of language

Not designed for a specific language, programming model or object model[6]

- Should act as a compilation target for all kinds of higher-level machine languages
- Some Examples for languages that can be used at the time of writing are: *TODO*

TODO

2.2.3.E Compact

Wasm bytecode representation format should be compact. Smaller binary files are easier and faster to transmit especially in web contexts [6]. Also smaller files can be loaded into memory faster at runtime, which might lead to a slightly faster execution overall, but this is more of a speculation.

TODO

2.2.3.F Modular

Programs consist of smaller modules, which allows modules to be "transmitted, cached and consumed separatly" [6].

TODO

2.2.3.G Other features of WebAssembly

This section lists some of Wasm's other noteworthy features. These are not directly related to this work, however they provide a better overview over Wasm's potential usecases and applications.

- **Goal: Well-defined**: Wasm is designed in such a way that it is "easy to reason about informally and formally" [6].
- **Goal: Open**: *TODO*

2.2.3.G Other features of WebAssembly

- **Goal: Efficient**: Wasm bytecode is efficient to read and parse, regardless of whether AOT or JIT compilation or interpretation is used at runtime [6].
- **Goal: Parallelizable**:
  Working with Wasm bytecode should be easily parallelizable. This applies to all steps: decoding, validation, compilation. This property allows for a faster startup time.
- **Goal: Streamable**:
  This goal is especially important for the web. It should be possible to parse Wasm code while it is still being streamed/received. On the web data can be transferred in separate blocks called chunks. Wasm bytecode allows a Wasm runtime to decode, validate and compile a chunk before the full bytecode has arrived. This reduces startup time for Wasm applications especially on the web.
- **Determinism**: Indeterminism has only 3 sources: host functions, float NaNs, growing memory/tables
- **Backwards-compatibility**: *TODO*

TODO

TODO

- **Migratability/Relocatability**: Running Wasm instances can be serialized and migrated to another computer system. However for this to be easy the Wasm runtime should only execute Wasm code through interpretation.

2.2.4 Challenges & Limitations

This section deals with common challenges and limitations of Wasm in non-web contexts.

TODO

2.2.5 WebAssembly System Interface (WASI)

TODO

2.2.6 WebAssembly Component Model

TODO

2.3 Plugin systems

TODO

3 Requirement analysis for plugin systems (20 pages)

3.1 Definition of requirements

TODO: Define key requirements that are important for plugin systems of any kind, also discuss the requirements for text editors specifically

- Extensibility for new features/interfaces (relativ)
  - Can the API be changed easily?
- Plugin Safety/Isolation
  - Checklist for safety to find out what is the worst case:
  - full access to the underlying hardware e.g. root privileges on os
  - full access to current user profile
  - access to peripherals e.g. network, connected disks, keyboard
- Plugin portability
  - Scores: not portable, portable but very hard (e.g. an entire VM is necessary), portable with runtime, portable without runtime (basically impossible, fat binaries for multi-architecture?)
- Language interoperability for plugin development (which/how many languages are supported)
  - Advantages:
    - More accessibility for developers without knowledge of a singular specific language.
  - Scores: a domain-specific language (custom language), multiple programming languages (JS, Python), a compilation target (JVM), no restrictions (machine code)
  - Some languages can be embedded reasonably well into others (e.g. JS in C)
- Plugin performance
  - Guess based on existing benchmarks (no time to write custom benchmarks)
  - Overhead of context switches between plugin system and plugins might be important?
- Plugin size
  - Guess based on existing benchmarks (no time to write custom benchmarks)

3.1 Definition of requirements

3.2 Market analysis of existing plugin systems

TODO: What is this section about?

TODO: Why is a market analysis important for the work of this paper?

3.2.1 Overview of chosen projects

TODO: How and why are these projects chosen?

- VSCode (versatile text editor)
- IntelliJ-family (IDE)
- Zed (text editor with Wasm plugin system, no windows support)
- ZelliJ (terminal multiplexer, has a Wasm plugin system)
- DLL-based plugins (e.g. FL studio)

3.2.2 Evaluation of key requirements

TODO: define a scale for rating each requirement

TODO: explain the methodology for evaluation: e.g. analysis of code, documentation, papers?

3.2.2.A Zed

TODO

3.2.2.B VSCode

TODO

3.2.2.C IntellJ-based IDEs

TODO

3.2.2.D Eclipse

TODO

3.2.2.E Microsoft flight simulator

TODO

3.2.2.F Zellij

Zellij is a terminal workspace (similar to a terminal multiplexer). It is used to manage and organize many different terminal instances inside one terminal emulator process.

3.2.2.F Zellij

Similary commomly known terminal multiplexers are Tmux, xterm or Windows Terminal on Windows.

- plugin system to allow users to add new features
- plugin system is not very mature https://zellij.dev/documentation/plugin-system-status
- Wasm for plugins, however only Rust is supported
- Permission system

TODO

3.2.2.G Extism

Extism is a cross-language framework for embedding WebAssembly code into a project. It is originally written in Rust and provides bindings (Host SDKs) and shims (Plugin Development Kits: PDKs) for many different languages. *TODO*

3.2.3 Summary

TODO: Present findings in a table TODO: Which other technologies might also be interesting? Which ones were left out?

4 WebAssembly for plugin systems (20 pages)

TODO: Present basic idea of running Wasm code for each plugin inside a Wasm runtime

4.1 Overview of basic plugin system architecture

4.2 Evaluation of requirements

4.3 Evaluation of interface-specific requirements

It is not possible to evaluate all requirements for WebAssembly. WebAssembly as a technology is often too unrestrictive and thus the decision of whether a requirement is fulfilled often comes down to the host- and plugin-language and whether a common interface definition between them exists.

To illustrate this point, consider a scenario in which a host system is written in JavaScript. When this host system wants to call a Wasm function it serializes the arguments, which might consist of complex JavaScript types, to JSON strings. Then it passes these JSON strings to the Wasm plugin. A plugin written in JavaScript itself will be able to easily parse the JSON string given to it, however a plugin written in C first has to get a system in place to parse and convert JavaScript types to equivalent C types.

TODO: Wasm interfacing is still an unsolved problem. There are many different solutions, of which some will be evalued separatly here

4.3.1 WebAssembly without a standardized interface

TODO

4.3.2 WebAssembly + WebAssembly System Interface

TODO

4.3.3 WebAssembly + WebAssembly Component Model

TODO

4.3 Evaluation of interface-specific requirements

4.3.4 WebAssembly + WebAssembly System Interface + WebAssembly Component Model

TODO

4.3.5 WebAssembly + custom serialization format (JSON, XML, Protobuf)

TODO

4.4 Summarized evaluation for WebAssembly

TODO: Show all WebAssembly configuration in a table with all requirements as columns

5 Proof of concept: Implementing a WebAssembly plugin system for a text editor (10 pages)

TODO: Provide context of helix text editor

5.1 Requirements

TODO: Weight the requirements for plugin systems and optionally add new requirements for this project

5.1.1 Functional requirements

5.1.2 Non-functional requirements

5.2 System Architecture

TODO: Choose appropriate Wasm technologies based on the previous findings and document how they are used together to build a working system

5.3 Implementation

TODO: Give brief overview over code structure TODO: Technical challenges and how they were addressed TODO: What optimizations were made?

5.4 Evaluation & Results

TODO: Reevaluate the key requirements for plugin systems TODO: Provide measurements for memory/performance impact (there are no real reference points) TODO: Summarize findings & challenges

Standard plugin definieren (z.b. textsuche für performance) Graph mit x geladenen Standard plugins für memory/performance

6 Results & Discussion (2 pages)

7 Outlook (2 page)

Bibliography

[1] Andrew S. Tanenbaum, *Structured Computer Organization*. [Online]. Available: https://csc-knu.github.io/sys-prog/books/Andrew%20S.%20Tanenbaum%20-%20Structured%20Computer%20Organization.pdf

[2] "Bringing the Web up to Speed with WebAssembly."

[3] "WebAssembly in Avionics."

[4] M. N. Hoque and K. A. Harras, "WebAssembly for Edge Computing: Potential and Challenges," 2022.

[5] "Potential of WebAssembly for Embedded Systems." [Online]. Available: https://arxiv.org/html/2405.09213v1

[6] WebAssembly Community Group and Andreas Rossberg (editor), "WebAssembly Core Specification." [Online]. Available: https://webassembly.github.io/spec/core