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

1.1 Motivation & problem statement 1

1.2 Research question 1

1.3 Method 1

2 Fundamentals (15 pages) 2

2.1 Instrucion set architectures 2

2.2 WebAssembly 2

2.2.1 Overview 2

2.2.2 Design goals 4

2.2.3 Challenges & Limitations 6

2.2.4 WebAssembly System Interface (WASI) 6

2.2.5 WebAssembly Component Model 6

2.3 Plugin systems 6

2.4 Rust 6

3 Requirement analysis for plugin systems (20 pages) 7

3.1 Definition of requirements 7

3.2 Market analysis of existing plugin systems 7

3.2.1 Overview of chosen projects 7

3.2.2 Evaluation of key requirements 7

3.2.3 Summary 8

4 WebAssembly for plugin systems (20 pages) 9

4.1 Overview of basic plugin system architecture 9

4.2 Evaluation of requirements 9

4.3 Evaluation of interface-specific requirements 9

4.4 Summarized evaluation for WebAssembly 10

5 Proof of concept: Implementing a WebAssembly plugin system for a text editor (10 pages) 11

5.1 Requirements 11
5.1.1 Functional requirements 11
5.1.2 Non-functional requirements 11
5.2 System Architecture 11
5.3 Implementation 11
5.4 Evaluation & Results 11
5.4.1 Conclusion 11

6 Results & Discussion (2 pages) 12

7 Outlook (1 page) 13

Bibliography 14

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 Method

2 Fundamentals (15 pages)

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

2.1 Instrucion set architectures

In the field of structured computer organization Tanenbaum defines a instruction 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 an ISA for a portable, time- and space-efficient code format, designed for but not limited to the web[2]. In the web it is used to execute code on a Wasm runtime, that is usually shipped with the client's internet browser.

What is special about Wasm is that it is a *virtual* ISA [2]. 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).

[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]

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

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[2]. 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]. However most compilers are still being actively worked on and improved over time.

[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***

2.2.1 Overview

2.2.2 Design goals

Wasm was designed with certain design goals in mind. This section presents the design goals according to the specification [2]. Each design goal is accompanied by related information from various papers and articles for a deeper understanding of each goal.

Some of these design goals are specific to the web, most notably Section 2.2.2.I or Section 2.2.2.F. However it turns out that most properties are generally desireable in non-web contexts too[4].

2.2.2.A Fast

- Fast Runtime
  - Parsable by a single-pass compiler: e.g. no backwards-jumps allowed (except loops), branches are referenced relativly and not absolutely (like jumps and addresses)
  - Minimal translation of opcodes: small set of basic opcodes, that exist on most architectures(***TODO: insert screenshot of opcode table*** (+ proposals for more additional instructions like SIMD, atomics, etc.)
  - Fast startup time (especially web): Runtime can start parsing a WebAssembly module that is still being downloaded. Goes hand in hand with Section 2.2.2.I
- WASM bytecode execute by WASM RT: AOT, JIT, interpreted
  - RT can switch between fast execution (AOT, JIT) or fast startup time (interpreted)

TODO

2.2.2.B Safe

- safe by default in sandbox
- 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

[4] e.g. portability and compactness for embedded systems [3] or portability, modularity and safety for distributed computing [4]

2.2.2.B Safe

- 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?*

TODO

2.2.2.C Well-defined

TODO

2.2.2.D Independence of hardware, language and platform

TODO

2.2.2.E Open

TODO

2.2.2.F Compact

TODO

2.2.2.G Modular

TODO

2.2.2.H Efficient

TODO

2.2.2.I Streamable

TODO

2.2.2.J Parallelizable

TODO

2.2.2.K Portable

- portable across architectures: Compilation target for higher level languages like C, C++, Rust

- embedding higher-level languages such as Python or JavaScript is also possible. In this case their runtimes are compiled to Wasm and shipped with the main Python/JS application.

TODO

2.2.2.L Other noteworthy features

- **Determinism**: Indeterminism has only 3 sources: host functions, float NaNs, growing memory/tables
- **Backwards-compatibility**: *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.

2.2.3 Challenges & Limitations

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

TODO

2.2.4 WebAssembly System Interface (WASI)

TODO

2.2.5 WebAssembly Component Model

TODO

2.3 Plugin systems

TODO

2.4 Rust

TODO: Is a Rust section necessary?

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

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?

- Zed (text editor with Wasm plugin system, no windows support)
- VSCode (text editor)
- IntelliJ-family (IDE)
- Eclipse (IDE)
- Microsoft flight simulator (has a Wasm plugin system)
- ZelliJ (terminal multiplexer, has a Wasm plugin system)
- Extism (generic Wasm plugin system library usable in many different languages)

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.C IntellJ-based IDEs

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. 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

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

5.4.1 Conclusion

TODO: Summarize findings & challenges

6 Results & Discussion (2 pages)

7 Outlook (1 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] WebAssembly Community Group and Andreas Rossberg (editor), "WebAssembly Core Specification." [Online]. Available: https://webassembly.github.io/spec/core

[3] [Online]. Available: https://arxiv.org/html/2405.09213v1

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