Goto table of contents

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

1.1 Motivation & problem statement

- Plugin systems are suitable architecture for highly individual software like text editors
- WASM is relativly modern bytecode that provides speed, safety, … by default.
- Can WAS

TODO: write motivation last

- WASM’s usecases are very limited -> it is still very new and evolving
- Plugin systems suffer from many issues: Security, interoperability, Portability, (Developer experience?)

1.2 Research question

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

1.3 Method (structure of this work)

- How will this work be structured?
- What technologies, languages and terms are used?

2 Fundamentals

This section introduces technical fundamentals for some technologies in this work.

2.1 Instruction set architectures

In the field of structured computer organization Tanenbaum defines the instruction set architecture (ISA) as a level in a multilayered computer system [1]. The ISA level defines a machine language (also sometimes referred to as machine code) 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

WebAssembly (Wasm) is virtual ISA for a portable, time- and space-efficient code format [2]. According to its specification it designed mostly for code execution on the Web, however it is not limited to the Web as an environement by design.

Even though one could design specific hardware that executes the WebAssembly ISA directly, this usually is not the case[1]. Instead WebAssembly calls itself a *virtual* ISA [2]. There is no common 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. We call this virtualized environemnt the **host environment** (used by the specification) or the **WebAssembly runtime** (used by most technical documentation).

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

2.2 WebAssembly

WebAssembly code can be written by hand, just as one could write ISA instructions (think of writing x86 machine code) by hand. Even though it is possible, it would not be efficient to write useful software in Wasm because the language is too simple. For example it provides only a handful of types: 1. Signed and unsigned integers, 2. floating point numbers, 3. a 128-bit vectorized number, 4. references to functions or 5. to host objects.

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. This is by design, as WebAssembly acts more as a compilation target for higher level languages[2]. Those higher level languages can then build upon Wasms basic types and implement their own memory layouts 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[2]. However most compilers are still being actively worked on and improved over time to support not only the basic WebAssembly specification but also extensions such as the Wasm Systems Interface and the Wasm Component Model, which are covered in Section 2.2.2 and Section 2.2.1 respectively.

Java bytecode is very similar to Wasm:

- Both define a virtual ISA
- Wasm is sandboxed by default, JVMs allow restrictions through a[3]
- Wasm is optimized to be parsed and compiled quickly. It allows for parsing/compilation to start while the bytecode is still being streamed/downloaded.
- Java uses GC, which may not be useful in time-critical software ***TODO: cite java usage in avionics***
- Wasm's machine language is more minimal and simpler, it thus is easier to compile a lot of languages to Wasm ***TODO: provide numbers for number of ops***

[2] `todo` provide examples for compilers that can target Wasm

[3] https://docs.oracle.com/javase/tutorial/essential/environment/security.html

2.2 WebAssembly

Wasm is designed with specific goals specifically suited for the web and efficiency in mind.

They include fast execution,

- Designed with specific goals in mind: …
  - 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
  - portable across architectures: Compilation target for higher level languages like C, C++, Rust[4] which is independant of the target systems' architecture.
  - safe by default in sandbox
- WASM bytecode execute by WASM RT: AOT, JIT, interpreted
- 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

TODO: source for 1.0 and date

- made for the web with 1.0 release in 2019
- no assumptions about execution enviroment are made
  - There are APIs for the web specifically (Javascript Embedding, Web Embedding)
  - 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?***
- Originally designed as: fast, safe, low-level bytecode for the web
- History and today's usage of WebAssembly

[4]In theory it can also contain other languages that require a runtime such as JS, Python

2.2 WebAssembly

Backwards compatibility: Bytecode compiled with previous compilers will always be valid.

- Examples:
  - Placeholders opcodes
  - Only one memory per module allowed, yet a memory index of 0 has to be included for some opcodes.
- New features go though 5 stages in their proposal process, before they are added to the specification. This allows for testing and for runtimes to implement proposals before they are standardized

TODO: Maybe include current browser support for stage 5 proposals from https://webassembly.org/features/

SOURCES: https://webassembly.org/docs/use-cases/, researched projects

- HPC inside the browser: Simulations, etc. `todo`
- Frontends written in compiled languages like C, C++, Rust (Leptos) by using frameworks. This still uses JS as a shim to access the DOM.
- Run untrusted code (TEXT EDITORS!! Plugins **do not** require complete access to FS, Network, etc.)
- Distributed systems where many nodes work together on a single computation. No Containerisation, Virtualization necessary. (Although WASM RT could be seen as a form of containerisation)
- Distributed systems: microservices, distributed computing

TODO: List most important proposals and why they are limiting factors for big projects

TODO: List all types here and show how a complex C type could be represented as a WASM type Here it must also be clear that it is completely language and compiler dependent how high level types are represented in the compiled WASM bytecode with its simple types. Here it becomes clear again that WASM only provides a very minimal execution environment and leaves all the responsibility and optimizations to compilers, very similar to tranditional assembly/machine code.

- Interfaces

2.2 WebAssembly

- WASM ↔ Environment: By default a WASM module cannot access its environment. It may export functions and import functions, provided by the host system through the WASM runtime. How to design a common interface that works for all WASM modules (regardless of their origin/compiler/ language) and all hosts and runtimes? Examples: Accessing a filesystem, Sending and receiving requests over a network.
- WASM ↔ WASM: How can two WASM modules, which may have been compiled by different compilers from different languages (Any combination of C, Rust, C++, JS, Python, …), communicate even though they have completely different memory layouts or one might even use GC? All languages allow for complex types made up of WASMs rather minimal type system Examples: Two WASM modules implement features, for example an efficient library for performing calculations in Rust and a Python module that uses that library and stores the results in a provided filesystem.

2.2.1 WebAssembly System Interface (WASI)

2.2.2 WebAssembly Component Model

- language-agnostic interfaces
- WebAssembly Interface Type (WIT) specification
- bindings can be generated for a lot of languages from a WIT definition

2.3 Plugin systems

- What are plugin systems?
- Why are plugin systems important?
- Where are plugin systems used?

2.4 Rust

- Short explaination of why Rust is used for this project:
- The text editor, for which a plugin system will be implemented as a proof of concept is written in Rust.

2.4 Rust

- However this work is not about any technology in particular, and instead it will focus more on WebAssembly, which may be used from pretty much any language where a runtime exists.

The text editor probably uses Rust for safety and performance reasons.

3 Requirement analysis for plugin systems

- Als Marktanalyse
- Analysis of related work and other projects/software
- Formal definition of requirements
- Projects
  - Text editors with WASM plugin systems: Zed
  - Common text editors with plugin systems: VSCode, IntelliJ, Eclipse
  - WASM plugin systems: Extism (cross-language plugin framework), maybe for audio processing?
  - Software with WASM plugin systems: Microsoft Flight Simulator, ZelliJ (terminal multiplexer)
- Works
  - Containerization with WASM in HPC: “Exploring the Use of WebAssembly in HPC” 10.1145/3572848.3577436
  - Structural and behavioural analysis of plugin components in Java: “Predictable Dynamic Plugin Systems” https://link.springer.com/chapter/10.1007/978-3-540-24721-0_9

3.1 Notes: Interesting projects

Zed Editor für Mac/Linux mit WASM Plugin System https://zed.dev/docs/extensions/developing-extensions https://github.com/zed-industries/zed/blob/94faf9dd56c494d369513e885fe1e08a95256bd3/crates/extension_api/wit/since_v0.2.0/http-client.wit

- WASM mit kompletter WIT Schnittstellendefinition (Generisches Typ/Funktions-basiertes System)pro Version
- Fokus auf Isolation von WASM Code
- Nur shim library für Rust vorhanden: zed_extension_api

Microsoft Flight Simulator

- Bietet unterschiedliche SDKs: WASM, JS, SimConnect SDK (?), SimVars (?)

ZelliJ Terminal multiplexer

- WASM mit WASI Unterstützung

3.1 Notes: Interesting projects

- API unabhängig von der Sprache mit Protobuf (Message-basiertes System, https://protobuf.dev/)
- Permission System gruppiert Events & Commands zusammen

Language Server Protocol

- wird oft als Ersatz für Plugin genutzt, um

4 WebAssembly for plugin systems

4.1 Overview

4.2 Choosing a plugin API

4.3 Safety

4.4 Performance

4.5 Summary

5 Implementing a WebAssembly plugin system for a text editor

5.1 Requirements

5.2 Design

5.3 Implementation

5.4 Example plugin development

5.5 Verification and validation

6 Results & Discussion

7 Outlook

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