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. First it covers the definition of an instruction set architecture. Then it gives an overview over WebAssembly, its features, challenges, limitations and extensions. Finally, plugin systems are explained as a software architecture model.

2.1 Instruction set architectures

In his book "Structured Computer Organization"[1] Tanenbaum defines an instruction set architecture (ISA) as a level in a multilayered computer system. The ISA level defines a machine language with a fixed set of instructions. 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. 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).

[1]There exist projects which have tried to execute Wasm directly. One example is the discontinued `wasmachine` project, which tried executing Wasm on FPGAs: https://github.com/piranna/wasmachine

2.2 WebAssembly

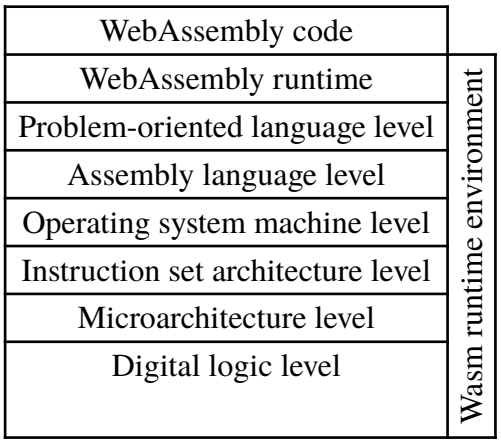

Figure 1: ***TODO: This figure still needs some improvements***. A multilevel computer system running Wasm code. Based on Figure 1-2 in [1]

If one considers a system running Wasm code as a multi-level computer system, the Wasm runtime can be modeled as a separate layer. Figure 1 shows a multi-level computer system based on Tanenbaum's model from [1]. Here each level is executed by logic implemented in the next lower level either through compilation or interpretation. The digital logic level itself only exists in the form of individual gates, consisting of transistors and tracks on the processors' chip. This level runs the next microarchitecture and ISA levels, which are also often implemented directly in hardware. The ISA level then provides a fixed set of instructions for higher levels to use. Operating systems build on top of this and provide another level for user space programs, which exist on the assembly language level. Then there are problem-oriented languages such as C, C++ or Rust, which are specifically made for humans to write code in[1].

One program written in a problem-oriented language is the Wasm runtime, which itself is a layer here. Its task it to interpret or (JIT-)compile higher-level Wasm code to lower-level problem-oriented or even the assembly language level. However for this work all layers starting with the Wasm runtime level until the digital logic level can be seen as a single hardware specific layer called the *Wasm runtime environment*.

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: Signed/unsigned integers,

2.2 WebAssembly

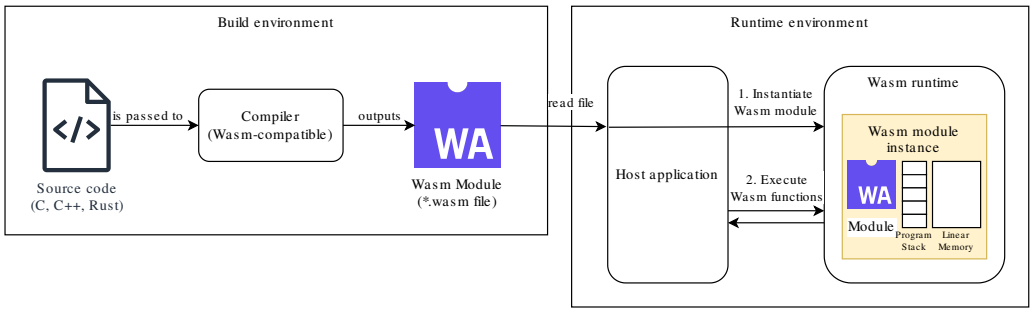

Figure 2: Flowchart for the creation and execution of a Wasm module from a higher-level language ***TODO: This figure is still too compilated***

floating point numbers, a 128-bit vectorized number and references to functions/ 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. 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 more of a compilation target for higher level languages[6]. Those higher level languages can then build upon Wasm's 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. Nowadays there are compilers for most popular languages already. Some of which are `clang` for C/C++, `rustc` for Rust or the official Zig and Haskell compilers. However most compilers are still being actively worked on and improved over time to support the latest Wasm proposals.

2.2.2 Execution model and lifecycle

Figure 2 shows the different stages a Wasm module goes through in its lifecycle. Its lifecycle starts with a developer writing source code. This source code can be written in an arbitrary programming language such as C, C++ or Rust. These languages are often used because they have compilers that support Wasm. The compiler then compiles this source code to a *Wasm module*. This step requires a compiler that support Wasm as a compilation target. Many modern compilers such as `clang` or `rustc` use the LLVM

project for optimization and code generation. These compilers compile source code to LLVM's intermediate representation (LLVM IR) and pass this to LLVM. LLVM can then perform optimizations and compile this IR to any compilation target, which can be selected by choosing a LLVM backend. One such LLVM backend targets Wasm. This way the Wasm-specific compiler logic can be implemented once within LLVM, and any compiler using LLVM can then target Wasm with minimal additional effort.

The compiled Wasm module exists in the form of a `.wasm` file. It is fully self-contained and unlike binaries it cannot depend on any dynamic libraries at runtime. It consists of different ordered sections, each with their own purpose: Some example sections contain function signatures, data segments, import- and export definition or actual Wasm instructions for each function.

The previously generated Wasm module can then be transferred to any target device or platform providing a Wasm runtime. That specific Wasm runtime is hardware-specific and not portable, unlike the Wasm module. The Wasm runtime is able to parse the received 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 executed. 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.

In a web context a server might provide this Wasm module to the client's browser, which usually comes with a Wasm runtime. Detailed information on the current status of Wasm support for internet browsers can be viewed at https://caniuse.com/wasm. A concrete example is Ebay using Wasm for their barcode scanner algorithm to achieve higher performance[2].

For distributed computing a compiled Wasm module could be distributed among multiple different nodes regardless of their platform and architecture… ***TODO***

[2]https://innovation.ebayinc.com/tech/engineering/webassembly-at-ebay-a-real-world-use-case/

2.2.3 Design goals

Wasm was designed with certain design goals in mind. This section presents the design goals relevant for this work according to the official Wasm specification [6]. 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 mostly optimized through the structure of the Wasm bytecode format, which is optimized for fast parsing and compiling of Wasm code. A Wasm module in its binary format consists of 11 different sections[3] that may only show up in a fixed order. Another example is the absence of backwards jump (except for the `loop` statement), which allows the use of faster one-pass compilers.

During runtime Wasm can also achieve near-native performances… ***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 runtimes for their specific needs and use cases. 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

TODO:

- ***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 environment 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, …***

[3] This section count excludes custom sections, which are optional and may only add additional information on top, such as debug information.

TODO: opcode table

Figure 3: 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.

- *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 environment, 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.*

2.2.3.C Portable

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

TODO:

- *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*

Independence of platform:

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

2.2.3.D Independence of language

TODO:

- *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*

2.2.3.E Compact

TODO:

- *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.*

2.2.3.F Modular

TODO:

- *Programs consist of smaller modules, which allows modules to be "transmitted, cached and consumed separably"*
- *Modules can also be combined/linked together at runtime*

2.2.3.G Other features

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

- **Well-defined**: *TODO: Wasm is designed in such a way that it is "easy to reason about informally and formally" [6].*
- **Open**: *TODO*
- **Efficient**: *TODO: Wasm bytecode is efficient to read and parse, regardless of whether AOT or JIT compilation or interpretation is used at runtime [6].*
- **Parallelizable**: *TODO: Working with Wasm bytecode should be easily parallelizable. This applies to all steps: decoding, validation, compilation. This property allows for a faster startup time.*
- **Streamable**: *TODO: 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**: *TODO: Indeterminism has only 3 sources: host functions, float NaNs, growing memory/tables*
- **Backwards-compatibility**: *TODO*

- **Migratability/Relocatability**: *TODO: 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 Criteria for plugin systems (20 pages)

To evaluate whether Wasm is a viable technology for versatile plugin systems, one must first understand what criteria make a plugin system good and versatile. This section will perform a technology comparison between several technologies and existing software projects. First, a set of criteria is defined. Then, appropriate technologies and software projects are selected to represent a spectrum of different plugin systems. Next, the technologies and projects are evaluated against the previously defined criteria. Finally, a technology comparison matrix is used to summarize and visualize the results.

3.1 Definition of criteria

In this section criteria for good plugin systems are defined. Each criterion will define a scale from 0 to 5, along with requirements for each score. This scale will be useful later to enable an objective evaluation and comparison of technologies and projects.

3.1.1 Performance

A computer's performance usually refers to the speed it is able to execute software at. Generally one wants every piece of code to run as fast as possible. However in some scenarios one might also choose other features such as less memory usage or dynamic typing over performance.

In the context of plugin systems, performance also refers to the speed at which software is executed. Here the most important software components are the host system, a plugin system and a plugin. In this case performance describes how quickly a host system can temporarily transfer execution to a plugin system, that then loads and invokes a plugin's entry point.

While performance can be measured quantitatively through benchmarks, in practice this is quite hard for plugin systems. To benchmark different plugin systems one would have to implement a variety of algorithms and scenarios for multiple plugin systems. Then one could measure the time each plugin system and plugin takes to execute.

Due to time-constraints and the wide spectrum of knowledge necessary for such a benchmark, this work will not utilize quantitative benchmarks to measure performance. Instead performance will be rated through educated guesses based on benchmarks and comparisons already available for chosen technologies. This section presents a rough

outline for the scores used, but the final score has to be determined for each technology individually.

TODO Performance scores

3.1.2 Plugin size

TODO: Maybe look at sizes of plugin systems as well? e.g. native has no overhead at all vs. JS needs an entire runtime with jit compiler

- Guess based on existing benchmarks (no time to write custom benchmarks)
- memory footprint might impact performance

3.1.3 Plugin isolation

Often times plugins contain foreign code. This is especially the case for text editors, where plugins are often downloaded from a central registry, also known as plugin/ extension marketplaces. Event though there might be checks in place to check for malicious contents, plugins are still foreign code.

- isolation is property how isolated plugin is from its host execution environment
- the interface between plugin and host plays a big role: only if it can be abused in unexpected ways, isolation is violated

Isolation is a property of plugins, that describes how isolated a plugin is from its outside execution environment.

0 – No isolation, required elevated privileges *TODO*

Worst case: Elevated privilege access to current system.

1 – No isolation ***TODO: Limited privilege access to current system, inherits host privileges***

Worst case: Full access to the current user's system and peripherals.

2 – In isolation with host application *TODO*

Worst case: Full access to host application.

3 – Partially sandboxed ***TODO: An attempt is made to restrict the plugin's access to the host system***

Worst case: Full access to host application.

4 – Fully sandboxed, dynamic interface *TODO*

Worst case: Access to parts of the host application not meant to be exposed due to a bug in the interface.

5 – Fully sandboxed, static interface The plugin runs fully sandboxed. It has no way of interacting with the host system, except for statically checked interfaces. Here statically checked interfaces refers to interfaces, that can be proven safe during compilation (or alternatively development) of the plugin system. One way to achieve this might be an interface definition in a common interface definition language. This restriction was chosen because it disallows plugin systems giving full access to parts of a host application without a proper interface definition.
Worst case: Indeterminable, a major bug in the sandboxing mechanism is required.

3.1.4 Plugin portability

Portability stems from the field of distributed systems. A high portability refers to software components, that can be moved from one distributed system A to another distributed system B without having to make any modifications[7]. This assumes that both systems A and B share a common interface for interaction to the software component[7]. In the context of plugin systems for text editors, portability can be interpreted in one of two different ways:

1. Every individual plugin is seen as a software component. This plugin is portable, if it can be loaded into two instances of the same text editor running on different platforms.
2. The entire plugin system itself is seen as a modular software component of a text editor. It is portable if it can be integrated into different text editors and run across different platforms.

This work considers only the first scenario, in which portability refers to each individual plugin.

0 – Not portable The plugin is not portable between different platforms. It is theoretically and practically impossible to run the plugin on different platforms.

1 – Theoretically portable The plugin is theoretically portable between different platforms. In practice this might be very complex and costly, e.g. having to run each plugin in its own dedicated virtual machine.

2,3,4 – Portable with a runtime The plugin is portable between different platforms, but it requires a runtime on the target platform. Because these runtimes can vary from one plugin system to another, a score range from 2 to 4 is specified here.

During evaluation the specific runtime has to be analyzed regarding its complexity and impact on the host system. A more lightweight runtime could also enable higher portability of the plugin system itself as described in Section 3.1.4.

5 – Portable by design The plugin is portable between different platforms without requiring a runtime on each host application. In practice this is very hard to achieve. Advanced technologies such as fat binaries, which are binaries that encapsulate compiled machine code for multiple different architectures, might be necessary.

Note that the most extreme scores 0 and 5 are very unlikely for any imaginable plugin system. 0 requires a plugin not to be portable at all, while 5 requires that a plugin is portable to different platforms and architectures which is very hard to implement on a technical level.

3.1.5 Extensibility for new features/interfaces (relative)

- Can the API be changed easily?
- 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)

TODO: define a scale for rating each criterion

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

3.2 Technology comparison of existing projects

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)
  - JavaScript based
- IntelliJ-family (IDE)
  - Java based
- Zed (text editor with Wasm plugin system, no windows support)
  - Wasm, but official interface only for Rust plugins?
- Zellij (terminal multiplexer, has a Wasm plugin system)
  - Wasm, but official interface only for Rust plugins?
- DLL-based plugins (e.g. FL studio)
  - Native code

3.2.1.A Zed

TODO

3.2.1.B VSCode

TODO

3.2.1.C IntelliJ-based IDEs

TODO

3.2.1.D 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. Similar commonly known terminal multiplexers are Tmux, xterm or the Windows Terminal.

- 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 Technology comparison of existing projects

3.2.2 Summary

TODO: Present findings in a table TODO: What could have been done better? TODO: Which other technologies and criteria 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

[7] Maarten Van Steen and Andrew S. Tanenbaum, *Distributed Systems: Principles and Paradigms*.