Proposal for CSE231 Course Project Multi-V-VM

Yiwei Yang Pooneh Safayenikoo Zheyuan Chen Tongze Wang
{yyang363,psafyen,zchen406,twang141}@ucsc.edu

October 2022

1 Introduction

MVVM is a high-performance double JIT VM from RiscV assembly or elf to wasm, with all Linux implemented. Here 'V' can be translated into multiple meanings: RiscV, Variable Level, etc. The second from WASM VM can be run on the browser side since the WASM has already supported outer SDK for developers to insert other logic of optimization or security. We are proposing a comprehensive system emulator that implements the RISC-V privileged ISA and supports interrupts, memory-mapped input and output devices, a soft memory management unit with separate instruction and data TLBs, and other features.

1.1 RiscV

RiscV is the newly proposed reduced instruction set by UC Berkeley that was supposed to be a student course project, but it quickly be famous and widely used by multiple companies among countries as their main ISA for National Defence [Sif22] and unique configuration for commercial needs [Ali22]. The decoding part for both RV32IMAFDV and RV64IMAFDV are different in instruction since the prior is 32bit length and with 32bit data, while the latter supports word(16bit), double word(32bit), and quadword(64bit) and the instruction are 64bit lengths. We need to implement them separately.

1.2 WebAssembly

WebAssembly is a new browser-supported feature that naturally supports the JIT at the ISA level. The browser has implemented a high-performance virtual machine called wasm3 [Shy22] to support gaming [vic22], virtual machine [Bel18] and video HVEC decoding[Bil22]. Because of the non-SSA design of the ISA and easy to prove with isolation [Aut18] and we've seen the application of isolation execution in serverless [She+20] and blockchain smart contract with the booming of the increasingly secured and full-fledged compiler.

2 Insight and Novelty

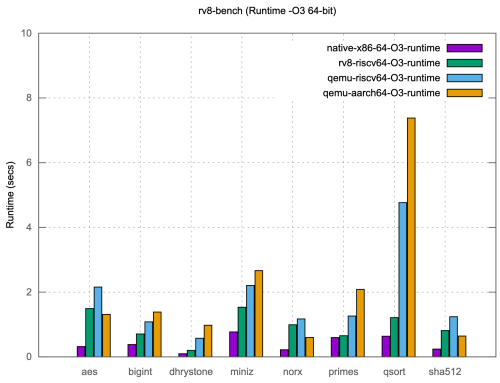

Figure 1: JITed rv8 compared with JITed Qemu

We all know that we can simply apply the LLVM JIT with Qemu JIT to do the job, however, the transpilation speed is slow when you combine all the architecture. Since the abstraction of the assembly and the machine will not always be zero cost, you are always gonna save more abstracted types of instruction or runtime memory types which makes the JIT slow. As rv8 benchmark [Cla20] listed, we've already gained 10x speed up compared to Qemu. We support double layer JITed, in which the context of transpilation has both the information of the first layer and also can utilize the wasitime [Hat22b] JIT to optimize for the machine code.

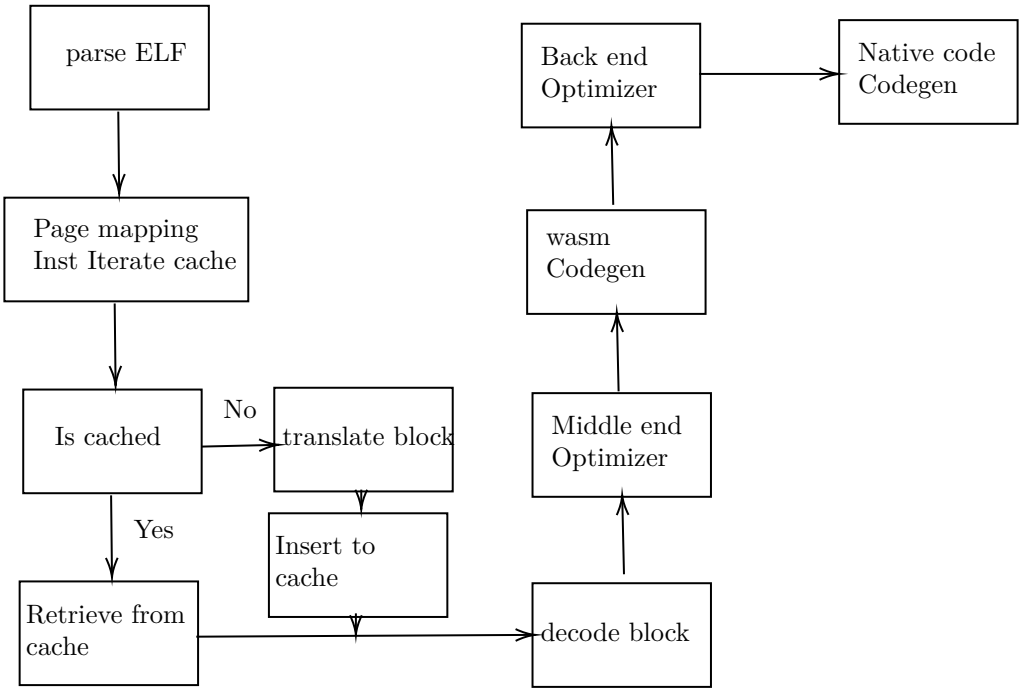

Figure 2: Diagram of the MVVM design doc

2.1 Comparison between RiscV and WASM ISA

2.1.1 Code/Data Separation

RiscV uses the same address space for code and data, while the running code of WASM does not even have a way to read/write itself. Simplify the implementation of the JIT compiler. If the code is self-modifying, then the JIT compiler needs to have the ability to detect changes and regenerate the target code, which requires a fairly complex implementation mechanism.

2.1.2 Static types and control flow constraints

WebAssembly is very "structural". The standard requires that all function calls, loops, jumps and value types follow specific structural constraints, e.g. passing two arguments to a function that takes three, jumping to a position in another function, performing a floating point add operation on two integers, etc. will result in compilation/validation failures; RISC-V has no such constraints, and the validity of instructions depends only on their own coding.

2.1.3 Machine Model

WebAssembly is a stack machine instruction set that will be JITed into the transpiled backend through a virtual machine, while RISC-V is a register machine instruction set.

2.1.4 Memory Management

Although WebAssembly and RISC-V both define an untyped, byte-addressable memory, there are some detailed differences between them; WebAssembly's memory is equivalent to a large array: the effective address starts at 0 and

expands continuously up to some program-defined initial value and can grow. RISC-V, on the other hand, uses virtual memory, using page tables to map addresses to physical memory.

2.1.5 Synchronization mechanism

A Turing-complete calculator requires at least one conditional branch instruction. Similarly, an instruction set architecture that supports multi-threaded synchronization requires at least one "atomic conditional branch" instruction. Such instructions are available under WebAssembly and under RISC-V, corresponding to the CAS model and the LL/SC model, respectively. LL/SC has stronger semantics than CAS, which suffers from intractable ABA issues, but LL/SC does not. This also means that it is much more difficult to simulate LL/SC on a CAS architecture than vice versa.

3 Proposed Architecture

3.1 Front end

We first parse the RiscV assembly. By getting the metadata in the elf, we know the linkage type, which architecture and etc. We then do the memory mapping to the WASM, during which we parse the instruction one by one using rust iterator and decoding using a code cache so that it will cache the block that has been translated before for better frontend optimization.

3.1.1 Normal Instruction

31 general-purpose registers and 31 floating point general-purpose registers(both have hard-wired zero) will be compiled completely to the WASM memory stack machine which we'll be LRU for register-stack mapping. As for function calls and the semantics of the stack frame and local variable, we will just translate how WASM interacts with the call convention.

3.1.2 RVV Instruction

We are supposed to support RVV Instructions, which add the `VLEN` global environment for one core. WASM has already equipped with basic SIMD [Aut22], which is just a mapping of SSE2. So the question is mapped to transpile from one SIMD to the other implementation of SIMD.

3.1.3 Multi Process Support

We will fully emulate how the process's order of accessing the variable to the semantic level. Since the semantic of RVWMO, the emulation of the weaker memory model will automatically be done on the stronger memory model, all we need to take care of is intrinsic of load_seq and store_release of taking care of atomic variable, which we can assign those to the atomic variable of a variable. Since the analysis through a wider range of searches for those atomic variables is hard, we try to map the semantics of RV as strong or conventional as possible.

3.1.4 Privilege Instruction

For different rings inside RiscV, we have to follow the semantics of the guest address and host address translation and we have to provide a software MMU here to get the translation right. In WASM, we think the same process shares nothing by addresses, thus the MMU for WASM is unnecessary.

3.2 Middle end

During translating the instruction, we have a middle end for optimizing from RiscV to WASM like the Solid-State Register Allocator [**ssrc**].

3.3 Back end

We need a WASM Runtime for LibC in the first place. Although there already exists a WASI implementation [Hat22a], we think we just need a lightweight Runtime for a better performance

3.3.1 LibC

There are 2 major Libc implementations out there, we are choosing `musl` for its simplicity. Let's give a comparison between `musl` and `sysv`. [Zha20]

1. `musl` is a much simple standard library compared to GlibC. When linked statically, binary files will be more size-efficient
2. `sysv` has more depth to get to call the real syscall stub, which is more heavy weighted.

3.3.2 Optimization

We can cast optimization over the wasitime-jit since they provide a context for code generation. We can have the following prospective optimization from WASM to native code.

1. Function Inlining
2. Common Expression Elimination

4 Conclusion

In the final report, we will provide a systematic review of different implementations of different RiscV VM from functionality and performance.

References

[Aut18] K framework Author. *K framework Proof of WebAssembly*. 2018. URL: `https://github.com/wasm3/wasm3`.

[Bel18] Fabrice Bellard. *Windows2000 on Virutal Machine*. 2018. URL: `https://bellard.org/jslinux/vm.html?url=https://bellard.org/jslinux/win2k.cfg&mem=192&graphic=1&w=1024&h=768`.

[Cla20] Michael Clark. *rv8: RiscV JIT emulator*. 2020. URL: `https://michaeljclark.github.io/`.

[She+20] Youren Shen et al. "Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 955–970. ISBN: 9781450371025. DOI: `10.1145/3373376.3378469`. URL: `https://doi.org/10.1145/3373376.3378469`.

[Zha20] Tom Zhao. *The comparison of GlibC between MUSL and SYSV*. 2020. URL: `https://blog.tomzhao.me/wp-content/uploads/2021/08/glibcmusl-system-call.pdf`.

[Ali22] Alibaba. *World's First Laptop with RISC-V Processor Now Available*. 2022. URL: `https://www.tomshardware.com/news/risc-v-laptop-world-first`.

[Aut22] K framework Author. *K framework Proof of WebAssembly*. 2022. URL: `https://github.com/WebAssembly/simd/blob/main/proposals/simd/SIMD.md`.

[Bil22] Bilibili. *Start the HEVC inside the Bilibili player*. 2022. URL: `https://zhuanlan-zhihu-com.translate.goog/p/514735090?_x_tr_sl=auto&_x_tr_tl=en&_x_tr_hl=en&_x_tr_pto=wapp`.

[Hat22a] Red Hat. *LibC implementation inside WASI*. 2022. URL: `https://github.com/WebAssembly/wasi-libc`.

[Hat22b] Red Hat. *WasiTime JIT Optimization tools for WASM*. 2022. URL: `https://crates.io/crates/wasmtime-jit`.

[Shy22] Volodymyr Shymanskyy. *The fastest WebAssembly interpreter, and the most universal runtime*. 2022. URL: `https://github.com/wasm3/wasm3`.

[Sif22] Sifive. *NASA Makes RISC-V the Go-to Ecosystem for Future Space Missions*. 2022. URL: `https://www.sifive.com/press/nasa-selects-sifive-and-makes-risc-v-the-go-to-ecosystem`.

[vic22] victoryang00. *A game called Senpai written in C++ compiled to WASM*. 2022. URL: `https://github.com/victoryang00/Ryan-teaching-winter-session`.