Multi-V Virtual Machine - A binary offloading paradigm onto the phone

Yiwei Yang Zheyuan Chen
Computer Science and Engineering
UC Santa Cruz
Room 381, 606 Engineering Loop, Santa Cruz, CA 95064
{yyang363,zchen406}@ucsc.edu

Abstract

In recent years, with the advancement of phones, we have many reasons to offload server binaries onto the phone with both performance[5], flexibility, and security. First, the computation resource is getting better, the latest iPhone chip A16 has been proven to have the same big core as the desktop M1 chip. Second, running docker on M1 mac is just using qemu-x86 to transpiling the x86 code that used to run some binaries that are unique to x86, despite rossetta2 is supporting Linux virtual machine, the offloading to phone just through the interface of web browser and a fine-tuned WebAssembly Virtual Machine can have much flexibility. Last, the WebAssembly proves to be safe [2][3]. In this project, We designed a double JIT virtual machine that can dynamically JIT from RISCV assembly to WebAssembly, and with help of WASI and WASMtime, we can JIT the libc linkage and compile it into native code. The Register mapping from the native code can guide the WebAssembly backend generation.

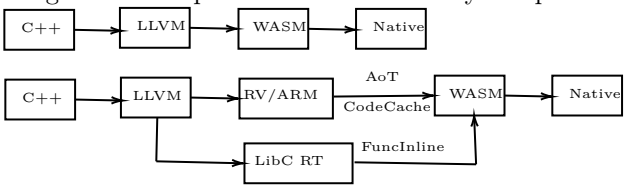
1. Introduction

1.1 Comparison with state-of-the-art compiler techniques

The WebAssembly s-expression files of the style [9] allow AoT compilation and radical optimization that takes the information from the first phase of the transpilation, etc. code caches, micro-operation caching & combining, and radical software reordering. We can make full use of the processor's information and do specific optimization like making fast libc calls, and making code alignment with the processor icache.

Normally, you can use the LLVM to compile to any language including WASM. our dissertation for that is this will require the implementation of memory order and CAS model in accordance with RV/X86/ARM when compiling a multithreading program, which may result in different semantics of the multithreading model, we can validate our JIT Virtual Machine cross-platform compatible in semantics by inserting barriers. Other than that, some SIMD intrinsic code doesn't support compilation to WASM if it's a fine-tuned version of code. Other than that, the WebAssembly normally takes 10x of code size that is not that code cache friendly, so it requires a hint of original semantics to tell the JIT Virtual machine that better tunes the icache performance.

Figure 1. Comparison of WebAssembly compilation



1.2 Comparison with state-of-the-art Virtual Machine for RV64

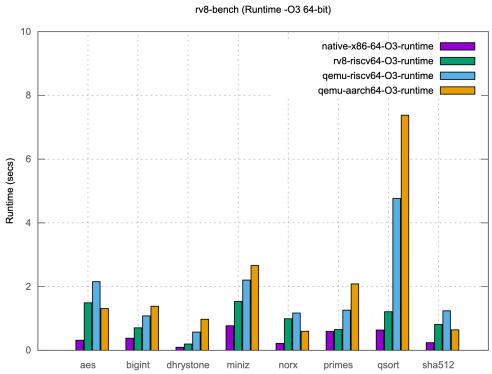
In this section, I will use 3 metrics, performance, flexibility and security to compare different approaches.

Here's a basic comparison of different of the performance of virtual machines.

Copyright © ACM [to be supplied]...$15.00
DOI: http://dx.doi.org/10.1145/(to come)

	performance	flexibility	security
qemu JIT	-	-	+
qemu	+	-	+
ckb-vm & valheim	-	+	+
rv8 & ria-jit	+	+	-
MVVM	+	++	+

Figure 2. The 3 metrics difference across the



The qemu has full emulation for all the CPU and outer devices even legacy devices. In 2018, the tiny code generator supported JIT that is it has an efficient bytecode IR with help of the virtual machine. Despite the functionality that it supports full operating system emulation on every device, it only supports user-level emulation from Linux to Linux on Linux machines.

Project ria-jit [8] and rv8 are implementations of RISCV JIT to X86. This kind of emulation can have the sophisticated mapping of the register but with limited functionality. As for flexibility, it can be compiled to WASM and run on the web browser. However, the JIT can make possible DeJITLeak[7].

Prototype riscv-jit-emulator [6] is a multi-backend emulator utilizing Execution Engine which has better flexibility and performance.

ckb-vm and valheim are full emulations for the CPU which is as slower as qemu.

1.3 Design of WebAssembly

2. Implementation

The code is now open-sourced in https://github.com/Multi-V-VM/MVVM/

2.1 Frontend Design

After the command line arguments are parsed, the guest executable is mapped into memory, as specified by the ELF header. The next step then is to initialize the guest environment. This includes setting up the guest environment as specified in the memory layout Section.

Some of the optimization systems also have to initialized, namely the code cache, the return address stack, the register file, and the context switcher.

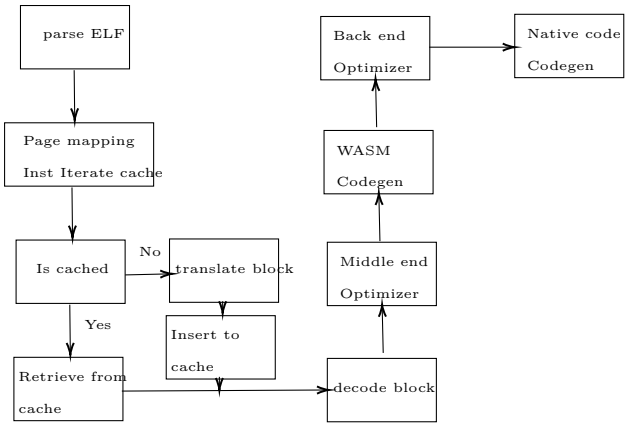


Figure 3. Diagram of the MVVM design doc

2.1.1 Instruction Code Iterator

To begin guest program translation and execution, the instruction iterator loop is started with the address next to be executed set to the entry point of the guest program.

After entering the loop (see figure 3), it is first checked if the block next to be executed has already been stored in the code cache.

If not, translation will begin by parsing the block's instructions, after which the pattern matcher(syscall, long jump/short jump, etc.) will find and replace any patterns that can be optimized. Following the translation of the parsed instructions, the block is saved in the code cache.

Next, it is checked if the last executed block ended in a jump that can be statically chained. If yes, the chainer is called to link the new block to the last one.

The last step is to switch to the guest context and start executing the block. Due to block chaining and the return address stack, multiple blocks might be executed before switching back to the host context, setting the address next to be executed, and returning to the transcode loop.

The loop will break if the guest program is terminated, after which the translator will terminate as well, returning the guest exit code.

2.1.2 Memory Layout

The memory layout of the WASM is different from the RISCV since it requires a faster virtual machine mapping to multiple backends. [10]. For global heap memory, we can simply map them into linear memory, with mapping of the memory load and storing to the translated correct memory instruction. The WASM language definition, is not a pure stack machine, because WASM locals can not be directly translated into stack machine SSA[4], it's better to translate using the callee saved register mapping to the locals. So we can just open as

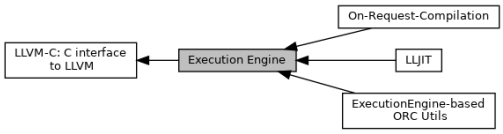
many const locals as possible for registers. The problem for getting another variable like static global variables in the data section or BSS section and loaded as a register, we can also map them as global WASM variables for semantic correctness.

2.1.3 Runtime CSR

Since there's a side effect of different rounding modes of RISCV floating point with exception or runtime architectural status in RISCV like mstatus register, vector instruction status register error, or interrupt vector status, we need a software emulation if there's no mapping in WASM. For final code generation, we can pattern-match the function to the backend architecture if needed. For instance, if the backend is ARM64, we can simply have a similar mapping of interrupt vector status.

2.2 Middleend Design

Double JIT requires memory, memory mapping, instruction, and transpiling command JIT to WASM with one iteration with another during the compilation. We have to design an execution engine of WASM, which is very similar to the LLVMOpaqueExecutionEngine[1]. This form of execution engine allows on-demand compilation, which means that one instruction in the frontend Instruction Iterator can be JITed through the Execution Engine right after its insertion into the module, just like [8] did.



To implement the WASM OpaqueExecutionEngine, we need to package a WASM builder and context, the resources are the WASM module, WASM function pointer.

for each instruction, the WASM module updates an instruction, and for each function, WASM module updates a function, then wasmtime the middle end is also able to execute a line by step. The middle layer should be wrapped in a wasmtime middle IR called cranelift.

2.3 Backend Design

We reused the wasm JIT implementation but will have a register mapping hint because the stack machine plus the const variable's reg allocation is not necessarily the same as the backend. Thus we need literally below 2 analyses. Our implementation pass is implemented over cranelift ARM64 backend.

2.4 Static register mapping

In order to achieve the best performance that is the same as before if we must decide how to make the best use of the limited number of host GPRs we have available.

Principally, we chose to statically map into as many registers as we could, without compromising on flexibility in handling instructions that require specific registers.

2.4.1 Register priority analysis

There are two main ways of determining the priority of registers when considering them as candidates for mapping.

It is, on the one hand, possible to assess the priority statically, by performing an analysis of the binary in question. Essentially, the hereby produced metric counts the number of times the register is used in the assembly instructions listed in the guest program and thus delivers an idea of how important each register is to this specific executable. We have built the tools required for this effort directly into the translator's analyzer function.

However, this approach does not take into account that a single instruction may be executed many times while the program is running. Accordingly, the other approach is to assess the registered priority dynamically by analyzing and profiling the execution of the testing program, thereby gaining an insight into how often each register is actually used during the execution. The translator is also capable of performing such an analysis.

A dynamic analysis, of course, delivers a largely more accurate idea of the priority of the registers in question but has the decided and obvious disadvantage that it cannot be performed without actually executing the binary.

3. Possible Experiment

We decided to run experiments first on the sanity test of test binaries introduced in riscv-jit-emulator [6] and then test the overall performance across the SPEC-2017 and eventually make tests around mobile binary offloading.

References

[1] L. Author. Llvm execution engine, 2022. URL https://llvm.org/doxygen/group___LLVMCExecutionEngine.html.

[2] W. K. framework Authors. Webassembly semantics, 2022. URL https://github.com/runtimeverification/wasm-semantics.

[3] R. Hjort. Formally verifying webassembly with kwasm, 2020. URL https://odr.chalmers.se/server/api/core/bitstreams/a06be182-a12e-46ce-94d3-cff7a5dc42ba/content.

[4] M. Keeter. Stack-based assembly easier to understand over register-based assembly, 2022. URL https://www.mattkeeter.com/blog/2022-10-04-ssra/.

[5] T. Kobayashi and K. Adachi. Probabilistic binary offloading for wireless powered mobile edge computing system. In 2020 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC), pages 1502–1506, 2020.

[6] J. Lifshay. Riscv jit emulator using llvmopaque-executionengine, 2020. URL https://github.com/programmerjake/riscv-jit-emulator/.

[7] Q. Qin, J. JiYang, F. Song, T. Chen, and X. Xing. Dejitleak: eliminating jit-induced timing side-channel leaks. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 872–884, 2022.

[8] G. Ria JIT Author from TUD. Dynamic binary translation (risc-v -> x86), 2020. URL https://github.com/ria-jit/ria-jit.

[9] D. Schuff. Webassembly aot prototype, 2020. URL https://github.com/dschuff/wasm-aot-prototype.

[10] A. Turner. Webassembly by example, 2020. URL https://github.com/torch2424/wasm-by-example/blob/master/examples/webassembly-linear-memory/demo/rust/src/lib.rs.