A Checkpoint/Restore Mechanism with Interoperability Among Distinctive WebAssembly Interpreters

Daigo Fujii
Future University Hakodate
Hokkaido, Japan
g2124038@fun.ac.jp

Katsuya Matsubara
Future University Hakodate
Hokkaido, Japan
matsu@fun.ac.jp

Yuki Nakata
SAKURA internet Inc.
Hokkaido, Japan
y-nakata@sakura.ad.jp

1 INTRODUCTION

WebAssembly (Wasm) has attracted attention because it acts as a lightweight virtual machine that absorbs platform heterogeneity, which is essential, especially for edge-cloud collaboration. In fact, several Wasm runtimes focused on edge computing, such as WasmEdge[1] for edge servers and Wasm Micro Runtime (WAMR)[2] for edge devices, exist. Furthermore, we can see the trend from the fact that the technology of VM migration, one of the essential features of cloud computing infrastructure, is proposed by Wasm-based cloud frameworks[1, 2].

Interpreters, known as the Wasm standard implementation, have the advantages of a quick startup, a small footprint, and sound portability. In contrast, the near-native performance of Wasm has been backed by the optimization techniques such as Just-in-TIme (JIT), Ahead-of-Time (AOT) compilation, and custom instruction set substitution known as 'fast' interpreters. Unfortunately, these code optimizations can get in the way of realizing the Wasm VM live migration, especially among heterogeneous runtimes. This preliminary study focuses on a Wasm VM checkpointing and restoring mechanism only for the following three interpreters, such as WasmEdge, WAMR, and Wasm3[3], excluding JIT and AOT implementations, to challenge technical issues on VM state conversion among the fast and standard interpreters.

2 INTEROPERABLE WASM VM STATE

WasmEdge adopts the standard interpreter that follows the Wasm specification, although WAMR and Wasm3 can be classified as fast interpreters. Most of the fast interpreters have been introduced primarily to improve stack transactions by transforming stack-based instructions to register-based ones. So, the code transformation influences the program counter, return values in the control stack, and the value stack more than the frame stack, memory, and globals; the Wasm core specification defines these classifications of VM state. Therefore, the VM checkpointing and restoring requires supplemental information to match them with each runtime's specific expressions.

Program Counter and Control Stack. The program counter is the instruction pointer of the next instruction to be executed, and the control stack manages the control flow as the instruction pointer that can be jumped to. The instruction pointers are absolute addresses, they can have different addresses on different machines and processes. Therefore, We implemented process that convert the instruction pointer to a relative address of Wasm bytecode. The instruction pointer in the standard interpreter points to Wasm code, enabling easy conversion. In the fast interpreter, it points to custom code, making conversion difficult. To make the conversion possible, we mapped the equivalent execution points of Wasm bytecode and custom bytecode. The mapping between Wasm code and custom code does not change the order of the code between Wasm code and custom code because the code conversion process of the high-speed interpreter converts Wasm code one instruction at a time. Thus, it is possible to map between execution points with equivalent program states.

Value Stack. The value stack has a different memory layout at each runtime and does not have type infomation of value stack. For example, in WAMR and Wasm3, all values inside the value stack are laid out, while WasmEdge is 0-padded in 128-bit units. Thus, it is difficult to convert value stacks of different memory layouts to each other. We achieved to convert different value stacks to each other by introducing a data structure that manages type infomation of value stack. Also, the value stack of the fast interpreter differs from that of the standard interpreter between equivalent code positions. Therefore, it is necessary to convert the value stack of the fast interpreter to the contents of the other. To get the value stack of the standard interpreter, traverse the Wasm code of the function at checkpointing. Since getting the value stack does not recalculate the value, but only collects the addresses where the precomputed value is stored, no loops or backward jumps occur during traversal, and the restoration can be done with little overhead.

[1] https://wasmedge.org/
[2] https://bytecodealliance.github.io/wamr.dev/
[3] https://github.com/wasm3/wasm3

REFERENCES

[1] Manuel Nieke, Lennart Almstedt, et al. 2021. Edgedancer: Secure Mobile WebAssembly Services on the Edge. In *Proc. of the 4th Int. Workshop on Edge Systems, Analytics and Networking (EdgeSys)*. 13–18.
[2] Mohammed Nurul-Hoque and Khaled A. Harras. 2021. Nomad: Cross-Platform Computational Offloading and Migration in Femtoclouds Using WebAssembly. In *Proc. of the 2021 IEEE Int. Conf. on Cloud Engineering (IC2E)*. 168–178.