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[5] for edge servers and Wasm Micro Runtime (WAMR)[1] 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[3, 4]. But, the lightweightness of Wasm execution has been backed by runtime optimization techniques such as Just-in-TIme (JIT) / Ahead-of-Time (AOT) native compilation and custom instruction set substitution, and unfortunately these can be obstacles for live migration among heterogeneous runtimes. This study focuses on implementing a VM checkpointing and restoring mechanism for multiple Wasm interpreters with unique custom instruction set substitution, known as 'fast' interpreters, such as WasmEdge, WAMR, and Wasm3[2].

2 TRANSFORMATION OF EXECUTION STATE

The fast interpreter has dynamic substitution from some of the standard Wasm bytecode with custom ones for performance optimization. WasmEdge adopts the standard interpreter that follows the Wasm specification, although WAMR and Wasm3 can be classified as fast interpreters. Therefore, an interoperable checkpoint/restore mechanism for VM migration among these interpreters needs to be created so that the program states coverage for the standard and fast interpreters and serializes it into a runtime-independent form.

The Wasm core specification defines the program state of Wasm. The program state of Wasm includes memory instances corresponding to heap memory, global instances corresponding to a global variable, a program counter representing the next code position to be executed, and three types of stacks that manage the execution. The stacks include a value stack that manages the result of the operation, a control stack that manages the code position that is the target of a branch or jump, and a frame stack that manages the call frames of active function calls. The standard interpreter is a straightforward implementation of the Wasm core specification, so can checkpoint/restore the state with serializing and deserializing.

The code positions must be able to match To implement checkpoint/restore between different bytecodes held by fast interpreters. The compilation of a fast interpreter can map Wasm code and the bytecode at many points, because of the nature of top-down Wasm code conversion. Although, there are changes such as one instruction generating multiple instructions or being omitted. For example, an instruction that only stores a fixed value on the value stack, such as the const instruction, is dropped and instead the value or address is referenced directly when compiling an instruction that uses the value of that instruction. The following two rules can establish a correspondence with the Wasm code and the runtime bytecode. First, when a single instruction generates several instructions, treat these multiple instructions as one. This means restricting the checkpointable execution point to the last instruction within the set of generated instructions. Second, if an instruction is skipped, map it to the position of the next generated instruction. If a code segment is dropped, it implies an associated program state affected by that instruction. The missing instruction can be mapped to the position of the next generated instruction since it does not alter the program state.

The contents of the value stack differ between Wasm code and runtime bytecode. This is because some instructions are omitted during the conversion to runtime bytecode, and consequently, the values these instructions would store in the value stack are also omitted. The premise is that the fast interpreter's bytecode is register-based; all instructions in Wasm code take arguments from the value stack and return the result of the operation to the value stack, but in the fast interpreter, the address to store the value argument and return value is held by the instruction's operand. Thus, instructions that return values independent of the result obtained at runtime, such as const and local.get, allocate an area for the value in a separate area from the value stack at compile time and pass that address to the operand. If the contents of the value stack are different, they must be converted to the contents of the value stack in the Wasm code before they can be checkpointed/restored. The solution is to use the value stack to construct the value stack information in Wasm at compile time in the fast interpreter. The value stack in a fast interpreter has insufficient content due to missing values. Yet, by maintaining this information at all code positions, the content of this address can be retrieved and checked at runtime. For restore, it is necessary to restore the value stack of Wasm without the missing values in the value stack of the fast interpreter. For this purpose, evaluation information is also added. The evaluation information refers to the values stored in the value stack of the fast interpreter at runtime. The instruction details that the fast interpreter omits and the instruction type information defined in the Wasm core specification provide this information.

REFERENCES

[1] Bytecode Alliance. Retrieved 12 Jul 2024. WebAssembly Micro Runtime. https://bytecodealliance.github.io/wamr.dev/.
[2] Wasm3 Labs. Retrieved 12 Jul 2024. Wasm3. https://github.com/wasm3/wasm3.
[3] Manuel Nieke, Lennart Almstedt, and Rüdiger Kapitza. 2021. Edgedancer: Secure Mobile WebAssembly Services on the Edge. In *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*. 13–18.
[4] Mohammed Nurul-Hoque and Khaled A. Harras. 2021. Nomad: Cross-Platform Computational Offloading and Migration in Femtoclouds Using WebAssembly. In *2021 IEEE International Conference on Cloud Engineering (IC2E)*. 168–178.
[5] Second State. Retrieved 12 Jul 2024. WasmEdge. https://wasmedge.org/.