MVVM: A WASI empowered remote process resuming light weight virtual machine

Yiwei Yang
University of California, Santa Cruz
Santa Cruz, California
yyang363@ucsc.edu

Brian Zhao
University of California, Santa Cruz
Santa Cruz, California
bwzhao@ucsc.edu

ABSTRACT

Checkpoint/Restore In Userspace (CRIU) has emerged as a prominent solution for live migration of Linux processes, offering a wide range of applications such as load balancing, fault tolerance, and resource management. However, the current implementation of CRIU is limited to supporting migration within the same Linux kernel version and architecture. This constraint substantially restricts its applicability in increasingly diverse and heterogeneous computing environments.

This thesis presents MVVM (Migratable Velocity Virtual Machine), a novel framework built upon WebAssembly System Interface (WASI) and WebAssembly (Wasm) that extends the capabilities of CRIU-based migration to encompass different kernels and architectures. MVVM leverages the platform-agnostic nature of WebAssembly to facilitate process migration across heterogeneous environments. The underlying idea is to make any applications play around on any device.

The proposed MVVM framework introduces a translation layer including code and memory, enabling seamless migration of process state information between different kernels like iOS, Linux, OSX, and architectures like X86, and Arm. It achieves this by converting the process state into an intermediate WebAssembly representation, which is then restored on the target system. Furthermore, MVVM integrates an optimized algorithm for incremental checkpointing and restoring, which minimizes migration downtime and reduces network overhead.

We evaluate the performance and effectiveness of MVVM by conducting extensive experiments in diverse simulated computing environments. Our results demonstrate that MVVM significantly expands the applicability of CRIU-based migration while maintaining acceptable performance overheads, paving the way for more edge computing stuff.

1 INTRODUCTION

The topic of migration in the scenario of heterogeneous binary offloading and remote resuming becomes a big problem. Say you a migrating the current process somewhere that has more power like the newest and greatest chip or operating system that has a better jitted or AoTed native library code. Compared with prior work CRIU, we support heterogeneous ISA, heterogeneous kernel.

Seamless migration in the context of heterogeneous device offloading and remote resuming poses several challenges. Some of the key challenges include:

Heterogeneous ISA (Instruction Set Architecture): Migrating processes across different instruction sets requires careful handling, as the target architecture may have different instruction encodings, register configurations, and memory layouts. One solution to this problem is to employ WAMR's dynamic code JIT+memory JIT IR over webassembly techniques to convert the source code or intermediate representation to the target ISA.

Heterogeneous Kernel: Different operating system kernels may have different system call interfaces, internal data structures, and resource management policies. Wasi supports the system call in a musl manner that supports most of the underlying modern operating systems.

Different versions of native libraries: The migration process may involve moving from an environment with one version of a native library to another environment with a different version. In some cases, this can lead to incompatibilities or performance issues.

State synchronization and Performance tradeoffs: During the migration process, maintaining input or process consistency between the source and target environments is crucial. We have a selection for maintaining an input restart or process state restart whenever which part is faster. We target the LLVM Memory JIT state.

Security and privacy: Migrating processes across different environments can expose sensitive data and potentially introduce new security vulnerabilities. To address these concerns, data encryption and secure communication channels like TLS can be employed during the migration process to protect sensitive information.

In summary, using Wasm+Wasi to do seamless migration in heterogeneous environments is a complex problem that requires addressing various challenges. Solutions such as re-JIT, compatibility layers, containerization, state synchronization, and security mechanisms can help ensure a successful migration process.

2 BACKGROUND

WebAssembly (Wasm) is a binary instruction format designed as a low-level virtual machine that runs code at near-native speed. It is a platform-independent format initially created to enable high-performance applications in web browsers. However, its potential extends beyond the browser, allowing for its use in other environments, such as IoT devices, edge computing, and server-side applications. Wasm supports a variety of higher-level programming languages, including C, C++, Rust, and more.

WebAssembly System Interface (WASI) is a standardized system interface for WebAssembly modules. WASI aims to provide a consistent and secure API for Wasm applications to access system resources, such as file systems, network sockets, and system clocks. By defining a standardized interface, WASI allows Wasm modules to be portable across different platforms and operating systems.

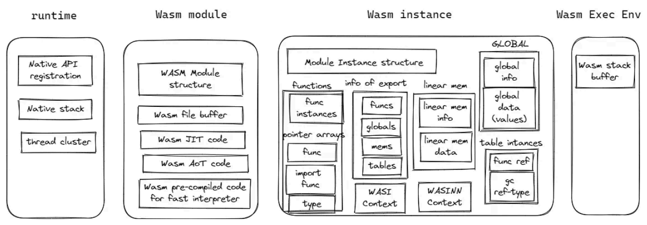


Figure 1: The running model in WAMR

3 DESIGNS

We build our PoCs on the wasm-micro-runtime(WAMR) project, available at https://github.com/Multi-V-VM/wasm-micro-runtime/tree/main, is an open-source WebAssembly (Wasm) runtime specifically designed for resource-constrained environments, such as IoT devices, edge computing, and embedded systems. This lightweight runtime enables developers to run WebAssembly applications efficiently and securely on various platforms with limited resources. The running model can be illustrated as below:

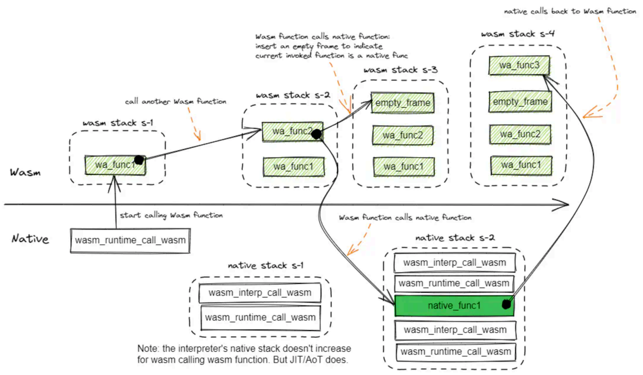
In the wasm-micro-runtime (WAMR) project, there are different build configurations to enable various features and execution modes. The following options are related to the WebAssembly (Wasm) execution modes:

(1) WASM_ENABLE_JIT: Just-In-Time (JIT) compilation is an execution mode where Wasm bytecode is compiled into native machine code at runtime using LLVM_JIT, right before execution. This approach allows for faster execution since the code is compiled and optimized specifically for the target system. However, it increases memory usage and startup time due to the compilation process. Enabling this option includes the JIT compiler in the WAMR build.
(2) WASM_ENABLE_AOT: Ahead-Of-Time (AOT) compilation is an execution mode where Wasm bytecode is compiled into native machine code before runtime, typically during the build process or when the application is installed. This approach results in faster startup time and improved runtime performance since the code has already been compiled and optimized. However, the compiled code may be larger and less portable across different systems. Enabling this option includes the AOT compiler in the WAMR build.
(3) WASM_ENABLE_FAST_JIT: Fast Just-In-Time (Fast JIT) compilation is a lighter version of the JIT compilation mode based on ASMJIT though the memory is applying LLVM_JIT.

Each of these execution modes has its advantages and trade-offs, depending on factors such as startup time, runtime performance, memory usage, and code portability. Depending on the specific requirements of the target environment and application, we choose to operate on LLVM_JIT.

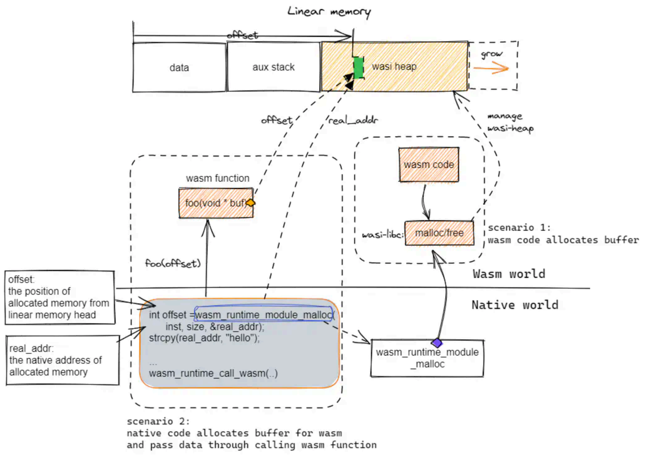
3.1 Heaps and stacks to migrate

WAMR uses two primary stacks: the native stack and the Wasm stack. The native stack is used for the execution of native code, including the WAMR runtime itself and native functions. The Wasm stack is used for the execution of WebAssembly code, including local variables and operand stack.


native calls back to Wasm function
Wasm function calls native function:
insert an empty frame to indicate
current invoked function is a native func
wasm stack s-4
wasm stack s-3
call another Wasm function
wasm stack s-2
wasm stack s-1
wa_func3
empty_frame
wa_func2
wa_func1
Wasm
start calling Wasm function
Wasm function calls native function
Native
wasm_runtime_call_wasm
native stack s-1
wasm_interp_call_wasm
native stack s-2
native_func1

Note: the interpreter's native stack doesn't increase for wasm calling wasm function. But JIT/AoT does.

Figure 2: The stack illustration in WAMR



offset: the position of allocated memory from linear memory head

real_addr: the native address of allocated memory

```
int offset =wasm_runtime_module_malloc(
      inst, size, &real_addr);
strcpy(real_addr, "hello");

...
wasm_runtime_call_wasm(..)
```

scenario 2:
native code allocates buffer for wasm
and pass data through calling wasm function

Figure 3: The heap illustration in WAMR

In addition to the stacks, WAMR also uses heap memory to store the linear memory of a Wasm module instance, which is used for global variables, dynamic memory allocations, and more.

3.2 Code Modification

(1) WASM App: A basic counter application written in C, which sleeps for one second and prints the counter value. To streamline the implementation initially, the program will call a native symbol (e.g., void __checkpoint(bool stop)) at a predefined iteration, such as the fifth one.
(2) WASM runtime: A simplified version of iwasm with the added __checkpoint native symbol.
(3) The streamlined iwasm version will support new command-line arguments, such as –restore or –restore-from-file, allowing iwasm to restore a module from its serialized state.
(4) Note that this proof-of-concept intentionally avoids any external calls other than __checkpoint.

4 PROPOSED EVALUATION

We will build three types of applications over Wasi and see the migration functionality and performance. We choose 3 kinds of applications: OLAP, inference and Graph Processing.

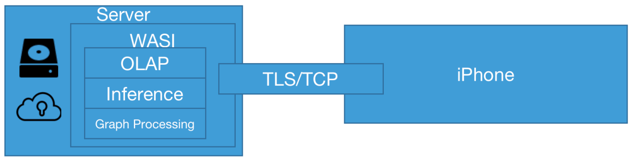


Figure 4: An example topology

5 RELATED WORK

We describe the two lines of this work from which Drywall takes inspiration:

The following points were discussed:

(1) [2] and [3] webassembly runtime explores an innovative approach to running a PHP/FAAS development server entirely within a web browser using WebAssembly (Wasm). This approach utilizes Wasm and WASI (WebAssembly System Interface) to create an environment where PHP/FAAS can be executed directly in the browser, eliminating the need for a separate server or backend infrastructure.

(2) [1] Checkpoint/Restore In Userspace (CRIU) is an open-source software project that enables the freezing and snapshotting of running processes on a Linux system and restoring them later on the same or another machine. This functionality is particularly useful for scenarios such as live migration, process-level fault tolerance, load balancing, and software debugging.

REFERENCES

[1] L. Foundation. Criu. URL https://github.com/checkpoint-restore/criu.
[2] S. Shillaker and P. Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. *arXiv preprint arXiv:2002.09344*, 2020.
[3] VMWare. Php runtime on webassembly. URL https://wasmlabs.dev/articles/php-dev-server-on-wasm/.