Bringing Together Cross-ISA Checkpoint/Restoration and AOT Compilation of WebAssembly Programs

Raiki Tamura
Kyoto University
Kyoto, Japan

Daisuke Kotani
Kyoto University
Kyoto, Japan
kotani@media.kyoto-u.ac.jp

Kazuyuki Shudo
Kyoto University
Kyoto, Japan
shudo@media.kyoto-u.ac.jp

Yasuo Okabe
Kyoto University
Kyoto, Japan
okabe@media.kyoto-u.ac.jp

Abstract

Cross-instruction set architecture (ISA) checkpoint/restoration is becoming increasingly important for live migration in heterogeneous computing environments, where applications need to move seamlessly between ARM, x86, and other processor architectures. While existing approaches either require compilation without Control-flow Integrity (CFI) or suffer from significant performance overhead through interpreter-based execution, this paper presents a novel approach that enables efficient cross-ISA migration using instrumentation during ahead-of-time (AOT) compilation. Our key insight is that on-stack replacement (OSR) enables cross-ISA checkpoint/restoration. OSR is a technique for JIT compilers, and we leverage it to transform between ISA-dependent machine states and ISA-independent WebAssembly states. Our other notable contribution is a technique enabling checkpointing without disabling modern CPU security features such as CFI. We implement the proposed techniques in Wanco, a WebAssembly AOT compiler supporting Linux on ARM-v8 and x86-64 architectures. Our evaluation demonstrates that Wanco achieves efficient cross-ISA migration compared to CRIU, a standard Linux process migration tool. Wanco reduces checkpoint time by 1.0–5.1 times and snapshot size by 1.1–25 times, while incurring an average execution-time overhead of 36 %.

CCS Concepts: • **Software and its engineering → Runtime environments**.

Keywords: WebAssembly, Virtual Machine, Compilers, Live Migration

ACM Reference Format:
Raiki Tamura, Daisuke Kotani, Kazuyuki Shudo, and Yasuo Okabe. 2025. Bringing Together Cross-ISA Checkpoint/Restoration and AOT Compilation of WebAssembly Programs. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3759426.3760985

1 Introduction

Checkpoint/restoration is a fundamental technique that enables saving and resuming application execution states. It plays a central role in modern data centers. A key use case is live migration, which moves running applications between hosts without service interruption and is essential for load balancing, system maintenance, and fault tolerance.

Historically, Intel x86 has been the de facto standard architecture in data centers. However, ARM and Power processors have recently gained significant traction due to their energy efficiency and performance benefits [8]. While this growing architectural diversity offers new opportunities for optimized resource usage, it also complicates application deployment and mobility.

In particular, live migration across different instruction set architectures (ISAs) is a significant challenge, as conventional hypervisor-based solutions save and restore CPU and memory states, which are inherently ISA-dependent. As a result, these approaches are inherently limited to homogeneous environments, where the source and destination machines share the same ISA. In heterogeneous systems—such as hybrid cloud platforms, edge computing setups, and mobile network infrastructures—cross-ISA checkpoint/restoration mechanisms are becoming increasingly critical to support seamless application migration and flexible system management.

Several approaches have been proposed to address cross-ISA checkpoint/restoration. One category of approaches modifies the operating system to support cross-architecture migration. Popcorn [9] provides a specialized operating system that supports task offloading between heterogeneous

Table 1. Comparison of virtual machines and native.

Runtime	Execution time	Portability	Source Language
Native	baseline	ISA-specific	Various (C, C++, Rust, etc.)[1]
Java	1.09–1.91× [12]	Portable across ISAs/OSes	Java, Kotlin, Scala
WebAssembly	1.55× [18]	Portable across ISAs/OSes	Various[1]

nodes by leveraging ISA-independent compiler intermediate representation (IR). Although Popcorn achieved user transparency and reasonable efficiency through substantial engineering effort, it imposes several practical limitations, including the requirement for application source code and the constraint that applications must be compiled without Control-flow Integrity (CFI) features such as Intel CET's Shadow Stack.

An alternative approach leverages language virtual machines, which define programs in an architecture-independent representation. This representation abstracts away ISA-specific details of machine state and allows execution through interpretation or just-in-time (JIT) compilation, enabling cross-platform portability. Virtual machines such as Java [13], C# [16], and WebAssembly [27] have long been recognized for their portability across operating systems and CPU architectures, making them natural candidates for cross-ISA live migration. Among these virtual machines, WebAssembly (Wasm) [15] represents a particularly promising solution for cross-ISA migration. It provides a compact stack-based program format designed for fast, safe execution in sandboxed environments. Unlike traditional virtual machines that target specific programming languages, WebAssembly is designed as a compilation target for systems programming languages such as C, C++, and Rust. Table 1 shows that WebAssembly achieves reasonable performance overhead while maintaining full portability across architectures.

Despite WebAssembly's growing adoption in cloud computing, edge environments, and web applications, research on checkpoint/restoration for WebAssembly remains limited. Existing approaches suffer from significant limitations: they either rely on slow interpreter-based execution models [11, 23] or incur substantial migration overhead due to bytecode-level program transformation [1], making them impractical for production environments where low latency and high performance are critical. The fundamental challenge lies in implementing checkpoint/restoration functionality in WebAssembly AOT compilers. While interpreters maintain an ISA-independent WebAssembly runtime state during execution, AOT compilers execute compiled machine code, where the runtime state becomes ISA-dependent, including processor registers and call stacks. This architectural difference makes it inherently challenging to capture and restore program state across different ISAs.

To address these limitations, this paper proposes techniques to implement cross-ISA checkpoint/restoration functionalities in AOT compilers for language virtual machines. Our approach employs machine code instrumentation during AOT compilation to enable checkpointing at specific program points without requiring application source code. Unlike existing approaches that require disabling CFI features, our instrumentation technique maintains compatibility with modern CPU security features by carefully preserving the program control-flow structure. By leveraging on-stack replacement (OSR), a technique in JIT compilers, our approach enables transformation between ISA-dependent machine states and ISA-independent WebAssembly states, overcoming the fundamental challenge of AOT compilation for cross-ISA migration.

We implemented these techniques in Wanco, a WebAssembly AOT compiler that currently supports Linux on ARM-v8 and x86-64 architectures, with plans for additional operating system support. Our evaluation compares Wanco's checkpoint-restoration performance with CRIU [2], a de facto standard tool for Linux process migration, demonstrating that Wanco achieves more efficient migration with 36 % overhead on average.

We make the following contributions.

1. Demonstrate that OSR enables cross-ISA checkpoint and restoration.
2. Propose a restoration technique that can be used with CFI.
3. Implement and evaluate a WebAssembly AOT compiler and show that it can perform live migration efficiently.

2 Background

2.1 WebAssembly

WebAssembly (Wasm) [15] is a portable binary instruction format designed for efficient and secure execution across platforms and architectures. Programs written in systems languages such as C, C++, and Rust can be compiled to Wasm, achieving near-native performance while retaining platform independence. Currently, however, Wasm supports only a restricted set of system-level features. For example, programs relying on inline assembly or stack unwinding mechanisms cannot be directly compiled at this time.

Wasm follows a stack-machine execution model: instructions implicitly push and pop values on a value stack rather

Table 2. WebAssembly runtimes and their performance [28].

Execution model	Runtime	Performance
Interpreter	Wasm3 [28]	baseline
JIT compiler	Wasmtime [7]	4.0×
	Wasmer singlepass [29]	2.5×
	Wasmer cranelift [29]	4.0×
	Wasmer LLVM [29]	7.6×
AOT compiler	WAVM [30]	9.2×

than explicitly naming registers. For example, the sequence `i32.const 1`, `i32.const 2`, `i32.add` pushes the constants 1 and 2 onto the stack, then pops them to compute their sum, leaving the result 3 on the stack.

A WebAssembly module consists of functions, global variables, linear memory, tables, and import/export definitions. Functions contain executable code and can call each other within the module. Global variables and function parameters have specific types (e.g., `i32`, `i64`, `f32`, `f64`), which are statically checked before execution. Linear memory serves as the program's untyped address space accessible through load/store instructions at arbitrary byte offsets. Tables store function references and enable indirect function calls. Imports and exports define the interface between the module and its execution environment, typically providing access to system functionality.

WebAssembly System Interface (WASI) [5] provides standardized APIs for system-level operations such as file system access and network operations, enabling WebAssembly modules to run as standalone applications outside web browsers. For example, `path_open` provides functionality similar to the POSIX `openat` system call in Linux.

2.2 Language Virtual Machines and Migratability

Nurul-Hoque et al. [23] argue that portability and migratability are key requirements for edge computing. Portability refers to the ability of applications and services to operate across heterogeneous environments without modification. This allows developers to deploy their applications seamlessly across different hardware platforms and operating systems. Migratability, on the other hand, denotes the capability to dynamically move running applications between different edge nodes or hosts. This feature is essential for optimizing resource utilization, ensuring service continuity in the event of node failures, and accommodating user mobility. Given that modern data centers are increasingly heterogeneous, portability and migratability are also critical in cloud computing contexts.

Popcorn utilizes LLVM IR, the intermediate representation of the LLVM compiler framework [21], to achieve portability. However, LLVM IR is not fully abstracted with respect to memory layout. The concrete layout of structs and arrays depends on target-specific type layout rules, which combine both ISA and ABI constraints, making it necessary to invest significant engineering effort when porting across operating systems and CPU architectures.

We argue that language virtual machines offer a more practical approach to achieving migratability compared to compiler intermediate representations. Unlike LLVM IR, which ties aspects of run-time state such as memory layout and calling conventions to specific ISAs and ABIs, language virtual machines abstract away these details, enabling execution across diverse runtime environments with different execution models. This versatility is particularly advantageous for computation offloading scenarios: on resource-constrained devices such as edge nodes or mobile platforms, interpreters avoid the compilation latency and memory footprint of JIT or AOT compilation, while cloud environments can exploit high-performance AOT compilers. Table 2 illustrates the performance trade-offs among execution models, showing indicative ratios reported in the Wasm3 repository [28]. These values should be viewed as approximate rather than definitive, since actual performance varies significantly with workloads and environments.

2.3 Checkpoint and Restore WebAssembly Runtimes

In this section, we explore two design options to checkpoint and restore WebAssembly programs with different trade-offs in terms of performance and implementation complexity.

Transforming WebAssembly Code. The first approach transforms WebAssembly programs by instrumenting instructions to enable pausing and resuming execution. Since WebAssembly does not natively support resuming execution from arbitrary points, this approach requires instrumenting the code with additional instructions to unwind and rewind the call stack, which introduces overhead in both performance and code size.

Asyncify [1] is a tool that performs such instrumentation to support features like coroutines. It works by transforming WebAssembly functions to handle asynchronous operations through stack unwinding and rewinding. Figure 1 shows a simple example of this transformation from Zakai [31]. The transformation introduces three execution states: `NORMAL`

```
function caller() {
  let x = foo();
  sleep(100);
  return x;
}
```

(a) Before transformation.

```
function caller() {
  let x;
  if (state == REWINDING) {
    // restore x
  }
  if (state == NORMAL) {
    x = foo();
  }
  if (state == NORMAL
    || callIndex >= 1) {
    sleep(100);
    if (state == UNWINDING) {
      // save callIndex=1 and x
      return;
    }
  }
  return x;
}
```

(b) After transformation.

Figure 1. Example of transformation with Asyncify [31].

for regular execution, UNWINDING when saving state and unwinding the call stack, and REWINDING when restoring state and rewinding to the resumption point. Each potentially blocking call is assigned a unique call index to identify the exact resumption location. When a blocking operation occurs (e.g., sleep(100)), Asyncify saves the current local variables and call index, then unwinds the stack by returning from all active functions. Upon resumption, it rewinds by re-entering the same functions, restoring the saved state, and skipping already-executed code using the call index.

To enable checkpoint/restoration with Asyncify, blocking function calls can be replaced with migration points. We evaluated the Asyncify-based checkpoint/restoration in Section 5 and found that it causes a substantial increase in code size due to the instrumentation required for checkpoint/restoration.

Modifying WebAssembly Runtimes. The second option is to modify implementations of WebAssembly runtimes to enable migration of WebAssembly programs. This method offers better performance because the runtime can directly access the program's execution state, eliminating the need to instrument the program with instructions to unwind and rewind the call stack. However, it requires additional engineering effort from runtime developers to implement the checkpoint–restoration mechanism in each runtime.

For interpreters, the implementation is relatively straightforward because interpreter-based runtimes typically maintain an explicit representation of the program's execution state [11, 23]. In contrast, JIT and AOT compilers present greater challenges, as their execution state resides in compiled and optimized native code, which utilizes CPU registers and native call stacks. Implementing checkpoint/restoration for these runtimes requires establishing a correspondence between WebAssembly execution state and native execution state, which not only complicates the migration process but also increases the implementation complexity.

3 Design

We propose a practical approach to cross-ISA process migration by performing binary transformation at the machine code level rather than at the Wasm level (Section 3.2). This design offers better performance and smaller code size compared to Wasm-level transformation. To handle application-level security features during migration, we introduce a restoration trampoline that safely restores execution state on the destination node (Section 3.3).

We implement this design in Wanco, an AOT compiler that compiles WebAssembly modules into migratable Linux executables. Figure 2 illustrates the system architecture of Wanco. The AOT compiler translates a WebAssembly module into a Linux object file. During compilation, it instruments two types of machine code: one to rewind and save the call stack, and another to check for a checkpoint request and capture the call stack if such a request is received. Wanco also provides runtime libraries that implement WASI functionalities and checkpoint-restore capabilities, such as serialization and deserialization of the runtime state.

3.1 Snapshots

Compiled programs can be paused upon receiving a checkpoint request, at which point a snapshot representing the program's current state is created. These snapshots encode the state of the WebAssembly runtime in a manner that is independent of the underlying operating system and CPU architecture, enabling program migration across heterogeneous platforms. To achieve this, Wanco adopts the runtime state representation defined by the WebAssembly specification interpreter [27], following the approach taken in previous studies [11, 23].

For WebAssembly programs, a snapshot captures the call stack, global variables, function tables, and linear memory. Each frame in the call stack includes the value stack, local variables, the function identifier, and the WebAssembly instruction offset from the beginning of the function.

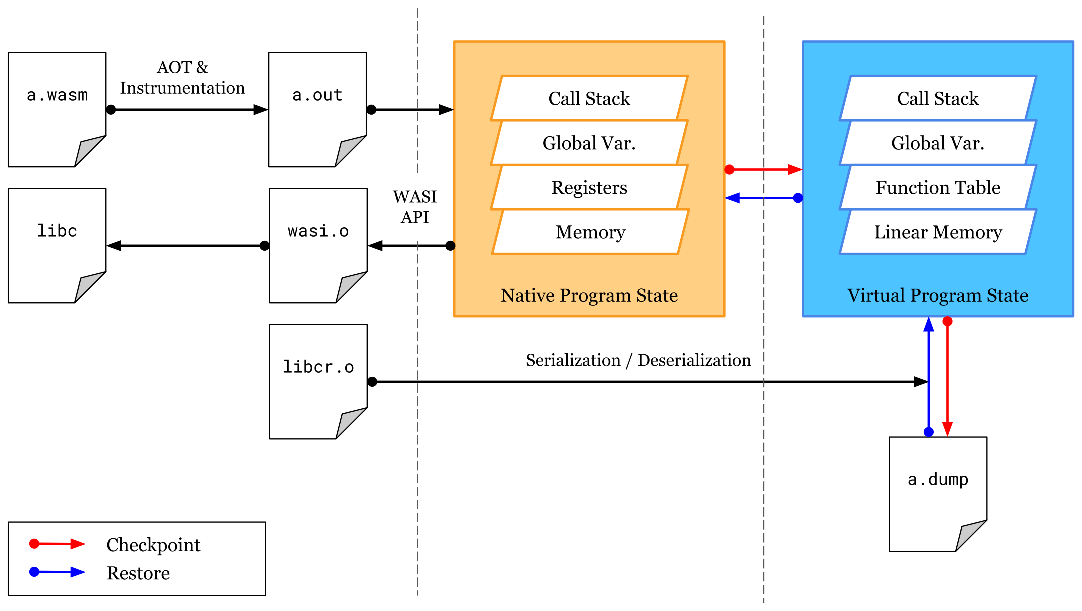

Figure 2. System architecture of Wanco.

3.2 Checkpoint and Restore

We discussed two approaches for migrating WebAssembly programs in Section 2.3. Among these, we adopt the second approach, which modifies WebAssembly runtimes to instrument machine code during AOT compilation. This approach is particularly suitable for AOT compilation scenarios, where peak performance takes precedence over startup time, unlike interpreter- or JIT-based execution models. In our approach, the AOT compiler inserts two kinds of machine code: migration points and code to save and reconstruct the runtime state.

Migration Points. Migration points are specific locations in a program where execution can be paused and resumed. The AOT compiler inserts them at the same locations as garbage collection (GC) safepoints: the beginning of functions and loop bodies, ensuring migration is possible even in infinite loops.

The AOT compiler generates machine code at each migration point, as illustrated in Figure 4, to check whether a checkpoint has been requested and, if so, to create a snapshot. Each migration point includes a conditional branch that checks whether a checkpoint request has been triggered. Since each migration point contains a conditional branch, increasing the number of migration points leads to higher performance overhead.

State Capture and Reconstruction. As we discussed in section 2.3, in AOT compilation, programs are compiled to register machines, requiring the ability to save the state during native execution as abstract WebAssembly states (snapshots) and conversely restore native execution from them.

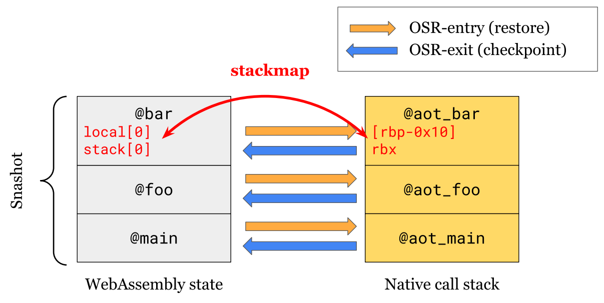

Figure 3. Checkpoint and Restore using OSR.

We propose an approach similar to on-stack replacement (OSR). OSR is a technique that allows switching between different execution modes during runtime, and we apply OSR's state transformation approach to our method. The key difference between OSR and our approach is that while OSR only transforms the state of the currently executing function, our method transforms the entire call stack (Figure 3). In our approach, OSR-exit (deoptimization) and OSR-entry (optimization) are performed for checkpoint and restoration, respectively. Both approaches share the common implementation detail of using stackmaps. Stackmaps record the locations of local variables in memory or registers and are emitted during AOT compilation.

3.3 Security Issues and Restoration Trampolines

To restore the execution of precompiled WebAssembly programs, the runtime must reconstruct the call stack at the

```
restore_point1:
if (state == CHECKPOINT) {
  checkpoint();
}
```

Figure 4. Code instrumented at migration points.

point of resumption. One straightforward approach to enable this reconstruction, which is used by Popcorn, is to allocate a memory region, write an image of the native call stack converted from the snapshot, and then switch the execution to use that native stack. Although this method is functional, it presents security challenges.

Modern CPUs implement security features to ensure the correctness of control flow. For example, Intel's Control-Flow Enforcement Technology (CET) [17] provides a mechanism called the Shadow Stack (Figure 6), which tracks function calls and verifies return addresses by comparing them between the call stack and the shadow stack. As a result, directly manipulating the native call stack can violate such mechanisms, rendering this approach incompatible with these security protections.

To address this issue, we introduce a new mechanism, called restoration trampolines. A restoration trampoline is inserted at the beginning of each function and dispatches execution to the appropriate program point upon restoration. Before jumping, it restores local variables and the value stack associated with that function.

Each migration point is generated as machine code similar to that illustrated in Figure 5. The runtime provides functions to read from the snapshot the WebAssembly instruction offset to jump to, as well as the local variables and value stack values to restore. It calls the `get_next_instruction_offset` API to obtain the WebAssembly program point to restore and then jumps to the corresponding location within the function based on this offset. Thus, if restoration targets a non-top function frame in the call stack, the trampoline jumps to just before the call instruction. Additionally, immediately before the jump, it calls the `get_next_local_value` API to load the values of local variables and the value stack.

4 Implementation

Wanco is forked from Wasker [26], a lightweight LLVM-based WebAssembly compiler written in Rust. Wasker only compiles WebAssembly modules to ELF object files and does not provide other functionalities such as WASI and checkpoint/restoration. For this reason, we do not include Wasker in Table 2, since it cannot directly produce standalone executables for performance comparison. To fill this gap, we implement these functionalities as a library in C++ and link them to the compiled WebAssembly module. Currently, Wanco supports compiling WebAssembly modules into executables for Linux on ARMv8 and x86-64 architectures. We plan to extend support to other platforms such as RISC-V and FreeBSD in the future.

We target Linux and utilize the POSIX signal `SIGUSR1` for checkpoint requests. Since POSIX signals are supported across various operating systems, this approach ensures portability when we extend support to platforms other than Linux in the future. The runtime's signal handler receives these signals and updates the global variable `state`, which represents the current execution or migration state and takes one of the values: `NORMAL`, `CHECKPOINT`, or `RESTORE`.

```
if (state == RESTORE) {
  offset = get_next_instruction_offset();
  switch (offset) {
    case OFFSET_ENTRY1:
      local1 = get_next_local_value();
      goto restore_point1;
    case OFFSET_ENTRY2:
      local1 = get_next_local_value();
      goto restore_point2;
    default:
      unreachable();
  }
}
```

Figure 5. Restoration trampoline.

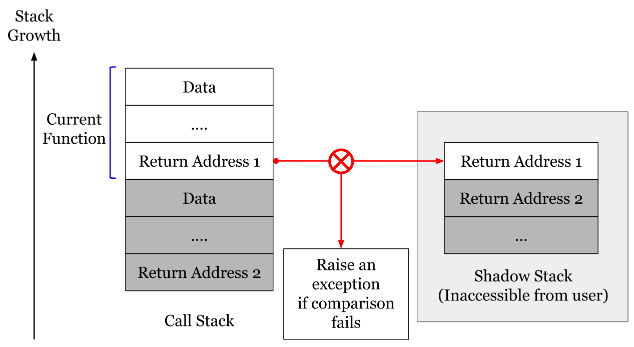

Figure 6. Intel CET's Shadow Stack.

4.1 Lowering WebAssembly to LLVM IR

We use LLVM as the compiler backend, and the AOT compiler translates WebAssembly modules into LLVM IR. First, WebAssembly functions are translated into LLVM functions. Since the types of values on the WebAssembly value stack at any program point can be determined at compile time, values on the value stack can be mapped to LLVM registers. WebAssembly global variables and function tables are translated into LLVM global variables. Function tables are represented as arrays of pointers to the translated functions. Linear memories are represented by LLVM global variables of pointer type and allocated at application startup. The special WebAssembly instruction `memory.grow`, which extends linear memory, is translated into a call to a native runtime library function that reallocates the memory if necessary.

Local variables are initially lowered using LLVM's `alloca` instruction, but because all WebAssembly locals are primitive types, LLVM's `mem2reg` pass usually promotes them to SSA registers. Thus, in optimized code, locals typically reside in registers and benefit from register allocation.

4.2 Migration Points and Compiler Optimizations

At each migration point, the runtime must be able to determine the WebAssembly instruction offset within the function. To maintain a consistent mapping between this offset and the underlying native execution state, compiler optimizations are constrained: instructions cannot be moved across migration points. In effect, these points serve as barriers to code motion, ensuring that the state observed at a migration point matches a well-defined position in the WebAssembly program.

Such barriers restrict optimizations like common subexpression elimination and loop-invariant code motion, which would otherwise move instructions beyond their original basic block. They also raise register pressure, since all live values on the WebAssembly stack must be preserved at migration points, sometimes introducing additional ϕ nodes or spills. These constraints apply only at migration points; between them LLVM can still perform its usual aggressive optimizations.

4.3 Multi-Architecture Support

Our implementation currently supports both `x86_64` and `ARMv8` architectures. Thanks to LLVM, most parts of the compiler and runtime required minimal modification to support multiple architectures. The backend code generation, IR lowering, and platform-specific optimizations are largely handled by LLVM itself.

However, certain low-level components still require ISA-specific handling. In particular, operations related to call stack unwinding and register preservation during migration must be implemented separately for each CPU architecture. These components are sensitive to calling conventions and the structure of the execution context, which vary across platforms. For each target architecture, we implemented approximately 200 lines of C++ and assembly code to handle these tasks. These implementations are modularized to ensure extensibility when adding support for additional architectures in the future.

Migration points are inserted at identical program locations across all ISAs. Therefore, the AOT compiler only needs to generate the code for saving and reconstructing the runtime state for its own architecture, without emitting target-specific code for other ISAs.

5 Evaluation

Instrumenting machine code during AOT compilation introduces both time and space overhead. Performance overhead arises from migration points and restoration trampolines, which are inserted at function entries and loop bodies. Since both contain conditional branches to check for checkpoint or restoration requests, they cause overhead even when checkpoint or restoration does not actually occur. Checkpoint and restoration themselves also incur costs. To create a checkpoint, the runtime converts the native call stack into a virtual call stack, serializes it, and writes it to a file. During restoration, the runtime reads the snapshot and reconstructs the call stack.

We evaluated our method using the benchmark programs listed in Table 4. We measured execution time, checkpoint and restoration time, and snapshot size to evaluate downtime, which includes the network transfer time of snapshots. Furthermore, we also measured code sizes because edge and embedded systems have limited memory and require distribution of AOT-compiled programs over networks. llama2.c [19] is a program that can run inference for a Large Language Model (LLM). We used the stories260K model from tinyllamas [20]. The Computer Language Benchmark Games [14] includes a set of very simple algorithmic problems. We used `nbody`, `binary-trees`, and `mandelbrot` from it. The GAP benchmark suite [10] contains a set of graph algorithms, from which we used `bc`, `bfs`, `cc`, `cc_sv`, `pr`, `pr_spmv`, and `sssp`. Note that `tc` crashed during checkpoint restoration and was excluded from the evaluation. For our evaluation, we used a synthetic Kronecker graph consisting of 2^{18} vertices. All evaluations were performed on the testbed described in Table 3, with each program run ten times.

5.1 Execution Time

We measured the execution time for each benchmark program and compared normal AOT compilation with AOT compilation with the checkpoint/restoration feature enabled. Additionally, we evaluated the performance of our AOT compiler against existing WebAssembly runtimes, WebAssembly Micro Runtime (WAMR) AOT [4] and WasmEdge AOT [3], to assess its competitiveness.

Figure 7 shows the ratio of execution time with the checkpoint/restoration (C/R) feature enabled to that without C/R for our AOT compiler. The performance overhead introduced by the C/R mechanism varies across different benchmarks. Execution time ratios (with C/R vs. without C/R) range from 0.89 to 1.82. The `nbody` benchmark showed a slight performance improvement with a ratio of 0.89, while the `binary-trees` benchmark experienced the highest overhead with a ratio of 1.82. On average, the performance overhead was approximately 36 %.

The execution time ratios of executables compiled with our compiler relative to those with WasmEdge ranged from

0.93 to 1.14, while those relative to WAMR ranged from 0.92 to 1.10. These results indicate that our AOT compiler achieves comparable performance to established WebAssembly runtimes.

5.2 Checkpoint and Restoration Time

We compared the time required for checkpoint and restoration operations in Wanco with those of CRIU. Checkpoint was performed at the halfway point of the average execution time, which was measured in advance. Figures 8 and 9 show the snapshot size and the migration time for Wanco and CRIU, respectively.

The checkpoint time varied significantly across programs. Wanco reduced checkpoint time by 0 % to 80.5 % compared to CRIU. The most significant improvement was observed in the `nbody` benchmark, where Wanco achieved a checkpoint time of 1.03 ms compared to CRIU's 5.28 ms. The restoration time also showed wide variation across benchmarks. In some cases, Wanco restored programs faster than CRIU, while in others it was slower. The restoration time of Wanco ranged from 0.54 to 41.96 times that of CRIU. The most significant improvement was observed in the `fannkuch-redux` benchmark, where Wanco restored the process in just 0.10 ms compared to CRIU's 4.00 ms. On the other hand, the highest overhead occurred in the `pr` benchmark, where Wanco's restoration time was 26.69 ms, whereas CRIU completed the same task in 14.37 ms.

5.3 Snapshot Size

Figure 8 shows that Wanco consistently produced smaller snapshots than CRIU, with size ratios ranging from 1.06 to 25.35. The most significant reduction was observed in the `fannkuch-redux` benchmark, where Wanco's snapshot size was only 128.3 KiB compared to CRIU's 3.18 MiB. Even in the worst case, `cc`, Wanco reduced the snapshot size to 68.13 MiB, which is 5.3 % smaller than the 71.95 MiB produced by CRIU.

This substantial reduction in snapshot size likely stems from Wanco's more precise state capture mechanism. While CRIU operates at the process level and captures the entire process memory space, including unused regions and runtime overhead, Wanco's WebAssembly-based approach captures only the essential application state required for restoration, which reduces unnecessary data inclusion.

5.4 Code Size

Figure 10 illustrates the code size comparison across 12 benchmark programs. We compare the code sizes of binaries generated by Wanco with those generated by WAMR and WasmEdge.

Our results show that WAMR produces binaries that are on average 69 % smaller than those generated by Wanco without C/R, with size ratios ranging from 0.28 (for `fannkuch-redux`) to 0.34 (for `llama2.c`). In contrast, WasmEdge generates binaries that are on average 45 % larger than Wanco without C/R, with size ratios varying from 1.05 (for `llama2.c`) to 1.78 (for `bfs`). This indicates that the binary size of Wanco without C/R is comparable to those produced by existing WebAssembly runtimes.

Table 3. Testbed Specifications.

Component	Specification
CPU	Intel Core i7-14700F
OS	Ubuntu 24.04.2 LTS
RAM	32 GB

Enabling C/R functionality in Wanco increases the code size by 116 % on average compared to the non-C/R version, with size ratios ranging from 1.99 (for `pr`) to 2.47 (for `llama2.c`). The most significant increase occurs when Wanco is combined with the Asyncify transformation, which instruments Wasm code to support stack unwinding and rewinding. Because Asyncify is required in approaches that implement checkpoint/restoration purely at the compiler level without modifying the runtime, the resulting binaries are on average 4.49 times larger than the standard Wanco. This ratio varies from 4.17 (for `fannkuch-redux`) to 4.68 (for `llama2.c`). While substantial, this increase is expected because the Asyncify transformation extensively instruments the code to enable state capture and restoration at the WebAssembly bytecode level.

These results demonstrate that Wanco with C/R strikes a practical balance between functionality and code size efficiency. It provides full checkpoint/restoration capabilities with a code size comparable to standard WebAssembly runtimes such as WasmEdge, while remaining significantly more compact than Asyncify-based solutions offering similar functionality.

6 Related Work

This section introduces related research on checkpoint and restoration of programs. Some of the works introduced in Sections 1 and 2 are also relevant here and are briefly revisited for comparison.

CRIU [2] is a process migration tool for Linux that enables freezing running Linux containers or applications and preserving them as snapshot files. These snapshots can be transferred to another Linux machine and restored, allowing applications to resume execution from their frozen state. CRIU has been integrated with container virtualization tools such as Docker [22], becoming the de facto standard for process migration in Linux environments.

Beyond process-level migration tools like CRIU, other approaches leverage compiler-level representations. For example, Barbalace et al. [9] propose Popcorn Linux, a system

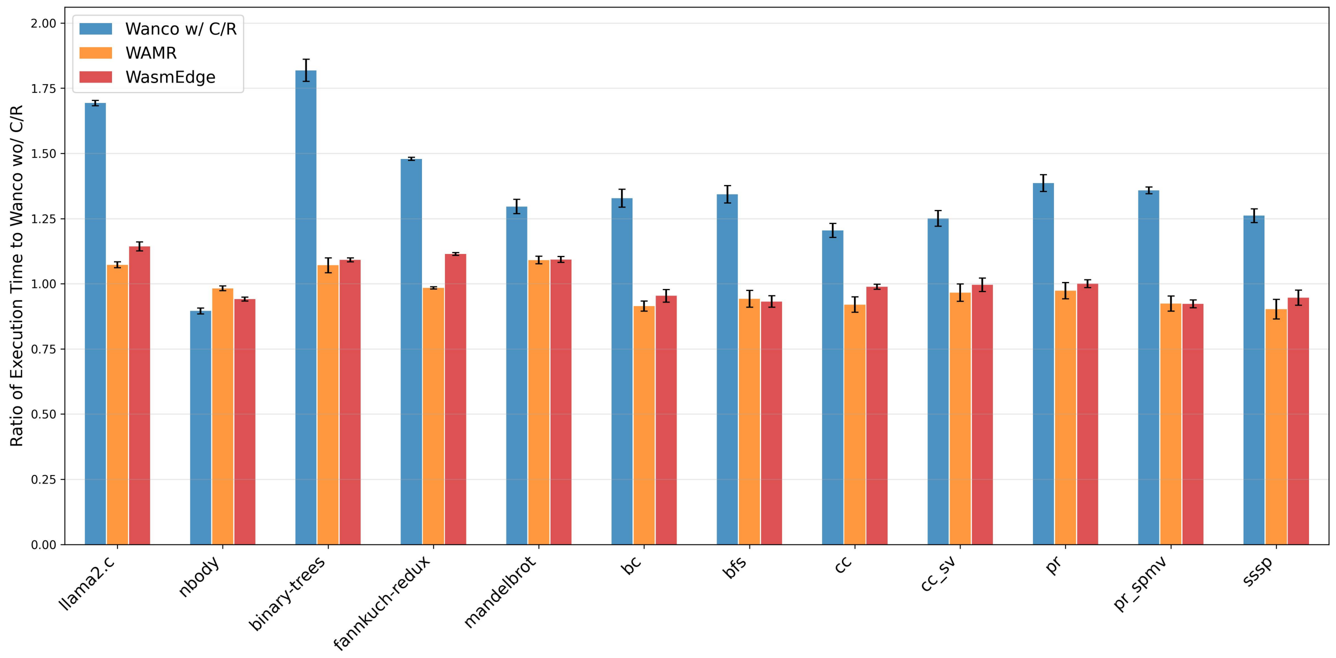

Figure 7. Execution time normalized to Wanco without C/R.

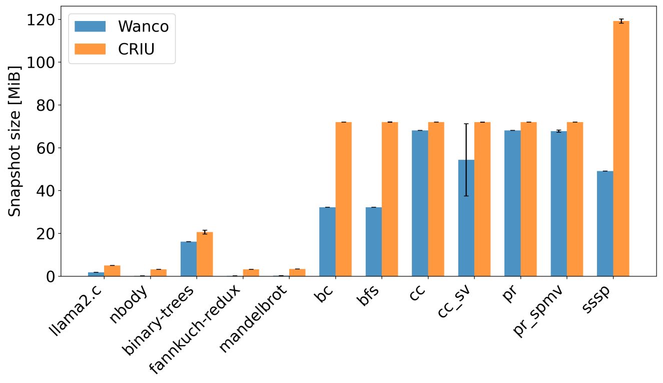

Figure 8. Snapshot size of Wanco and CRIU.

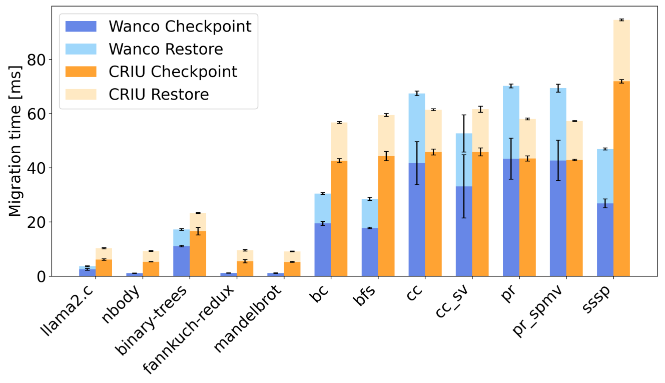

Figure 9. Migration time of Wanco and CRIU.

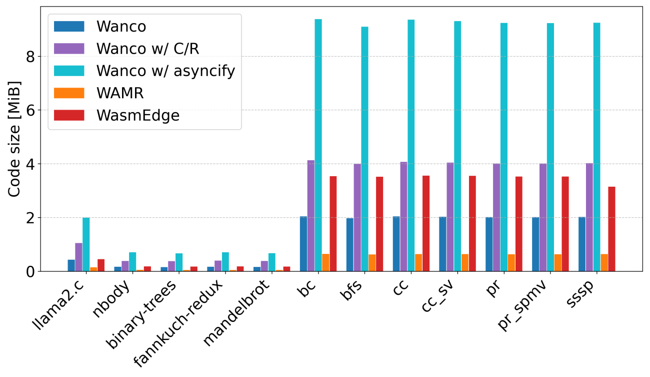

Figure 10. Comparison of code size.

Table 4. Benchmark Programs.

Benchmark	Program	Arguments
llama2.c [19]	`llama2.c`	`model.bin -n 0 -s 42`
CLBG [14]	`nbody`	`10000000`
	`binary-trees`	`18`
	`fannkuch-redux`	`11`
	`mandelbrot`	`1000`
GAPBS [10]	`bc`	`-g 18 -n 1`
	`bfs`	`-g 18 -n 1`
	`cc`	`-g 18 -n 1`
	`cc_sv`	`-g 18 -n 1`
	`pr`	`-g 18 -n 1`
	`pr_spmv`	`-g 18 -n 1`
	`sssp`	`-g 18 -n 1`
	`tc`	`-g 18 -n 1`

that utilizes LLVM IR, a compiler intermediate representation, for Linux process migration. Their approach enables transformation of call stack states across disparate ISAs, facilitating seamless migration in heterogeneous computing environments. However, their design targets Linux exclusively, limiting its portability to other operating systems.

Other studies focus on language virtual machines. Suezawa [25] and Shudo [24] independently proposed migration capabilities for the interpreter in the Sun Java Virtual Machine (JVM). Suezawa modified the interpreter, while Shudo implemented it as a plug-in to the JVM. Similarly, Agbaria et al. [6] modified the interpreter in the OCaml virtual machine to enable live migration. For WebAssembly, Nurul-Hoque et al. [23] and Fujii et al. [11] independently conducted related research, modifying WebAssembly interpreters to support migration.

Binaryen [1] is an open-source WebAssembly optimizer comprising multiple passes that improve code size and execution speed by transforming WebAssembly code. Its Asyncify pass instruments WebAssembly bytecode to enable pausing and resuming program execution by explicitly unwinding and rewinding the call stack. This technique supports asynchronous programming and facilitates state capture and restoration at the WebAssembly bytecode level.

7 Conclusion

In this study, we developed a WebAssembly AOT compiler and proposed implementation techniques for language virtual machines with checkpoint and restoration support. Our method utilizes the interpreter's state as a snapshot and converts the optimized code state into the interpreter's state when checkpoint or restoration is performed, by instrumenting machine code during AOT compilation. Although our approach incurs a 36 % execution time overhead on average, it achieves a significantly smaller code size compared to bytecode-level instrumentation such as Asyncify. Moreover, it provides practical migration performance with shorter downtime than CRIU, making it suitable for real-world use cases.

However, there are remaining challenges, such as the inability to save and restore external runtime states, including network and filesystem states.

References

[1] WebAssembly [n. d.]. *Binaryen*. WebAssembly. Retrieved April 26, 2025 from https://github.com/WebAssembly/binaryen

[2] CRIU [n. d.]. *CRIU Wiki*. CRIU. Retrieved November 21, 2024 from https://criu.org/Main_Page

[3] WasmEdge 2019. *WasmEdge*. WasmEdge. Retrieved May 13, 2025 from https://github.com/WasmEdge/WasmEdge

[4] Bytecode Alliance 2019. *WebAssembly Micro Runtime*. Bytecode Alliance. Retrieved May 13, 2025 from https://github.com/bytecodealliance/wasm-micro-runtime

[5] WebAssembly 2019. *WebAssembly System Interface*. WebAssembly. Retrieved June 3, 2025 from https://github.com/WebAssembly/WASI

[6] A. Agbaria and R. Friedman. 2002. Virtual machine based heterogeneous checkpointing. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS)*.

[7] Bytecode Alliance. [n. d.]. *wasmtime*. Bytecode Alliance. Retrieved June 6, 2025 from https://github.com/bytecodealliance/wasmtime

[8] Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. 2017. Breaking the Boundaries in Heterogeneous-ISA Datacenters. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 645–659.

[9] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. 2015. Popcorn: bridging the programmability gap in heterogeneous-ISA platforms. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys)*.

[10] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619* (2015).

[11] Daigo Fujii, Katsuya Matsubara, and Yuki Nakata. 2024. Stateful VM Migration Among Heterogeneous WebAssembly Runtimes for Efficient Edge-cloud Collaborations. In *Proceedings of the 7th International Workshop on Edge Systems, Analytics and Networking (EdgeSys)*. 19–24.

[12] Luca Gherardi, Davide Brugali, and Daniele Comotti. 2012. A java vs. c++ performance evaluation: a 3d modeling benchmark. In *Proceedings of the Third International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*. 161–172.

[13] James Gosling. 2000. *The Java language specification*. Addison-Wesley Professional.

[14] Isaac Gouy. [n. d.]. *The Computer Language Benchmarks Game*. Retrieved May 3, 2025 from https://benchmarksgame-team.pages.debian.net/benchmarksgame/

[15] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 185–200.

[16] Anders Hejlsberg, Mads Torgersen, Scott Wiltamuth, and Peter Golde. 2010. *C# Programming language*. Addison-Wesley Professional.

[17] Intel. 2025. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Order Number: 325462-087US.

[18] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. 2019. Not So Fast: Analyzing the Performance of WebAssembly vs.

Native Code. In *2019 USENIX Annual Technical Conference (USENIX ATC)*. 107–120.

[19] Andrej Karpathy. 2023. *llama2.c*. Retrieved October 12, 2024May 3, 2025 from https://github.com/karpathy/llama2.c

[20] Andrej Karpathy. 2023. *tinyllamas*. Retrieved May 2, 2025 from https://huggingface.co/karpathy/tinyllamas

[21] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. 75.

[22] Dirk Merkel. 2014. Docker: lightweight Linux containers for consistent development and deployment. *Linux J.* 2014, 239 (2014).

[23] Mohammed Nurul-Hoque and Khaled A. Harras. 2021. Nomad: Cross-Platform Computational Offloading and Migration in Femtoclouds Using WebAssembly. In *Proceedings of the 2021 IEEE International Conference on Cloud Engineering (IC2E)*. 168–178.

[24] Kazuyuki Shudo and Yoichi Muraoka. 2001. Asynchronous migration of execution context in Java Virtual Machines. *Future Generation Computer Systems* 18, 2 (2001), 225–233.

[25] Takashi Suezawa. 2000. Persistent execution state of a Java virtual machine. In *Proceedings of the ACM 2000 Conference on Java Grande*. 160–167.

[26] Soichiro Ueda, Ai Nozaki, Daisuke Kotani, and Yasuo Okabe. 2024. Mewz: Lightweight Execution Environment for WebAssembly with High Isolation and Portability using Unikernels.

[27] W3C. [n. d.]. WebAssembly Core Specification.

[28] Wasm3. [n. d.]. *Wasm3*. Wasm3. Retrieved June 6, 2025 from https://github.com/wasm3/wasm3

[29] Wasmer. [n. d.]. *wasmer*. Wasmer. Retrieved June 6, 2025 from https://github.com/wasmerio/wasmer

[30] WAVM. [n. d.]. *WAVM*. WAVM. Retrieved June 6, 2025 from https://github.com/WAVM/WAVM

[31] Alon Zakai. 2019. *Pause and Resume WebAssembly with Binaryen&apos;s Asyncify*. WebAssembly. Retrieved May 27, 2025 from https://kripken.github.io/blog/wasm/2019/07/16/asyncify.html

Received 2025-06-20; accepted 2025-07-28