Wasm-R3: Record-Reduce-Replay for Realistic and Standalone WebAssembly Benchmarks

DOEHYUN BAEK*, KAIST, South Korea
JAKOB GETZ*, University of Stuttgart, Germany
YUSUNG SIM, KAIST, South Korea
DANIEL LEHMANN, Google Germany GmbH, Germany
BEN L. TITZER, Carnegie Mellon University, USA
SUKYOUNG RYU, KAIST, South Korea
MICHAEL PRADEL, University of Stuttgart, Germany

WebAssembly (Wasm for short) brings a new, powerful capability to the web as well as Edge, IoT, and embedded systems. Wasm is a portable, compact binary code format with high performance and robust sandboxing properties. As Wasm applications grow in size and importance, the complex performance characteristics of diverse Wasm engines demand robust, representative benchmarks for proper tuning. Stopgap benchmark suites, such as PolyBenchC and libsodium, continue to be used in the literature, though they are known to be unrepresentative. Porting of more complex suites remains difficult because Wasm lacks many system APIs and extracting real-world Wasm benchmarks from the web is difficult due to complex host interactions. To address this challenge, we introduce *Wasm-R3*, the first record and replay technique for Wasm. Wasm-R3 transparently injects instrumentation into Wasm modules to *record* an execution trace from inside the module, then *reduces* the execution trace via several optimizations, and finally produces a *replay* module that is executable standalone without any host environment—on any engine. The benchmarks created by our approach are (i) realistic, because the approach records real-world web applications, (ii) faithful to the original execution, because the replay benchmark includes the unmodified original code, only adding emulation of host interactions, and (iii) standalone, because the replay benchmarks run on any engine. Applying Wasm-R3 to web-based Wasm applications in the wild demonstrates the correctness of our approach as well as the effectiveness of our optimizations, which reduce the recorded traces by 99.53% and the size of the replay benchmark by 9.98%. We release the resulting benchmark suite of 27 applications, called *Wasm-R3-Bench*, to the community, to inspire a new generation of realistic and standalone Wasm benchmarks.

CCS Concepts: • **Software and its engineering** → **Software maintenance tools**.

Additional Key Words and Phrases: WebAssembly, Benchmarking, record and replay

ACM Reference Format:
Doehyun Baek, Jakob Getz, Yusung Sim, Daniel Lehmann, Ben L. Titzer, Sukyoung Ryu, and Michael Pradel. 2024. Wasm-R3: Record-Reduce-Replay for Realistic and Standalone WebAssembly Benchmarks. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 347 (October 2024), 27 pages. https://doi.org/10.1145/3689787

*Both authors contributed equally to this research.

Authors' Contact Information: Doehyun Baek, doehyun.baek@kaist.ac.kr, KAIST, Daejeon, South Korea; Jakob Getz, jakob@getz.de, University of Stuttgart, Stuttgart, Germany; Yusung Sim, yusungsim@kaist.ac.kr, KAIST, Daejeon, South Korea; Daniel Lehmann, mail@dlehmann.eu, Google Germany GmbH, Munich, Germany; Ben L. Titzer, btitzer@andrew.cmu.edu, Carnegie Mellon University, Pittsburgh, USA; Sukyoung Ryu, sryu.cs@kaist.ac.kr, KAIST, Daejeon, South Korea; Michael Pradel, michael@binaervarianz.de, University of Stuttgart, Stuttgart, Germany.

ACM 2475-1421/2024/10-ART347
https://doi.org/10.1145/3689787

Table 1. A comparison of record and replay frameworks.

	PinPlay [43]	**JSBench [45]**	**Jalangi [50]**	**Wasm-R3 (this work)**
Cross-architecture replay		✓	✓	✓
Cross-language replay	✓			✓
Accurate replay	✓		✓	✓
Code format	native binaries	JavaScript	JavaScript	WebAssembly

1 Introduction

WebAssembly (Wasm) is a portable, low-level code format designed for compact representation and efficient sandboxed execution. It is primarily used as a compilation target for various source languages, including C/C++, Rust, and Kotlin, enabling new classes of software to be run in the browser. Its low-level instructions closely map to hardware instructions, achieving near-native performance with straightforward compilation techniques. Wasm augments the web platform, promising to speed up specific components of broader applications [27], as Wasm code often runs faster than JavaScript for numeric and memory-intensive tasks.

The complexity and diversity of Wasm engines demand robust, representative benchmarks for proper tuning. On the web, Wasm code is loaded dynamically from URLs or can be dynamically generated. Thus, code processing time, interpretation overhead, and JIT compilation to native code contribute to the overall application run time. To deliver fast startup time and high peak performance, all of today's web browsers employ multi-tier Wasm engines. For example, V8 and SpiderMonkey, the engines used in Chrome and Firefox respectively, use two compiler tiers [1, 10], while JavaScriptCore in Safari uses an interpreter and two compiler tiers. Non-web Wasm engines often also use multiple tiers, such as the Wizard Research Engine [52] which employs a new in-place interpreter design and a baseline compiler [53]. Multi-tier engines have complex performance characteristics and their tiering heuristics need to be tuned on realistic applications to ensure both startup speed and peak performance are maximized. Tuning these systems requires large, complex workloads that are representative of real-world applications. In the past, unrepresentative benchmarks, such as SunSpider for JavaScript [46], have misdirected engineering effort. In one instance, bad benchmarks led engineers into believing that their performance optimization resulted in a 13× improvement, when a representative benchmark showed only a 3× improvement [45].

Unfortunately, creating a sufficiently large set of representative Wasm workloads is challenging. One possible approach might be to port existing native applications to Wasm. However, the lack of standardized system APIs has made porting or recompiling large native applications difficult, as efforts like WASI [14] still lack basic facilities, such as signals, sockets, permissions, shared memory, and device APIs. Moreover, beyond the three major web engines, there are now several non-web production Wasm engines, which support disparate APIs or only subsets of WASI, making it difficult to have complex benchmarks with non-trivial system interaction. This deficiency in Wasm benchmarks compels researchers to either write their own benchmarks or use commonly used standard benchmark suites with few dependencies, such as PolyBenchC [44] and libsodium [17], which have already been shown to be unrepresentative of real-world applications [30]. Another approach might be to extract benchmarks from real-world web applications. However, Wasm-based web applications consist not only of Wasm modules, but also of a host environment that runs JavaScript code, interacts with the network, and interacts with the user. These properties make the web host environment difficult to emulate, which poses a problem for creating benchmark suites from representative web applications.

Outside of the Wasm context, creating good benchmarks is a long-standing challenge in many areas of systems. Efforts span virtual machines, operating systems, architecture, vision, and machine learning [7, 12, 26, 34, 41]. Key considerations are the size and runtime of benchmarks, ease of compiling and running them, licensing of the underlying source or binaries, diversity of the suite, representativeness of chosen benchmarks, and standardized measurement and reporting methodologies. For nascent and developing domains, writing new benchmarks makes sense, but for established domains with real-world usage, benchmarks should reflect actual applications to direct tuning efforts to benefit real-world usage. One such benchmark for Wasm is PSPDFKit [25], which measures the runtime of different actions of a PDF library. Yet, despite being well-crafted, this benchmark still depends on web APIs and requires significant effort to disentangle it to run on other engines. Moreover, creating and maintaining benchmarks like this requires significant manual engineering effort, so few examples exist so far.

If creating and curating benchmarks requires so much manual effort, why not automate the process? A promising approach is to automatically record and replay executions of real-world applications. Indeed, record and replay techniques for several systems and languages have been proposed, as shown in Table 1. However, each of these techniques lacks in a different dimension. PinPlay [43] supports recording of execution across multiple architectures, but cannot replay across different architectures. Language-specific efforts like JSBench [45] and Jalangi [50] are portable across CPUs and OSes, but only serve one language. Recording JavaScript is challenging; JSBench cannot record all memory loads, which means the execution at replay might diverge from the original execution [50]. Moreover, these techniques are not directly applicable to Wasm, and to the best of our knowledge, there currently is no record and replay technique for Wasm.

To address the lack of realistic benchmarks for Wasm, we present *Wasm-R3*, the first record and replay technique for Wasm that enables the creation of benchmarks from executions of real-world applications. Our key insight is that the design of Wasm modules enforces a clear separation between imported host functionality and the state and behavior inside a Wasm module, and that this is a natural boundary for encapsulating a benchmark. Our Wasm-R3 approach consists of three phases: record, reduce, and replay. To *record* an execution, the approach transparently injects instrumentation into Wasm modules and records all interactions with the environment. Because naïvely recording all interactions would result in an impractically large trace, Wasm-R3 *reduces* the trace via several optimizations. Finally, Wasm-R3 produces a *replay* benchmark that contains the unmodified code of the original Wasm module, but factors out the host environment and replaces it with a replay mechanism included directly in the replay benchmark.

Recording and replaying at the module boundary is akin to techniques for replaying native binaries at the system-call layer. Yet unlike native binary replay techniques [13, 42, 43], which often use memory-checkpointing techniques, the diverse host environments and engine offerings for Wasm demand a more general technique that works with an *uncooperative* host environment. The term "uncooperative" here means that the host environment does not provide any support for record and replay. Instead, Wasm-R3 works without any modifications of the host environment or the underlying Wasm engine, but instruments a Wasm module so that it records its own trace.

The benchmarks created with Wasm-R3 are portable, i.e., they work wherever Wasm runs. Since Wasm is gaining adoption across a broad range of contexts, such as Cloud [38], Edge [22], and IoT [35], Wasm-R3 must work across architectures, operating systems, runtimes, and host environments. While some record and replay techniques rely on support by the hardware [56], the operating system [18], or the language runtime system [49], the vast diversity of Wasm means that no one of these techniques can apply to all Wasm environments. Instead, Wasm-R3 produces self-contained, standalone Wasm modules that replay their execution faithfully not only on the engine used to create the benchmark, but on any engine. Moreover, the produced modules include the unmodified functions from the original Wasm module, only adding replay functions. Since the technique is primarily additive, the performance characteristics of the original functions are similar.

$$\begin{array}{rcl@{\qquad}rcl}
\textit{module} & ::= & \textit{function}^* \; \textit{global}^* \; \textit{start}^? \; \textit{table}^? \; \textit{memory}^? & \textit{import}, \textit{export} & ::= & \texttt{"name"} \\
\textit{function} & ::= & \textit{type}_{\textit{func}} \; (\textit{import} \mid \textit{code}) \; \textit{export}^* & \textit{code} & ::= & (\textbf{local} \; \textit{type}_{\textit{val}})^* \; \textit{instr}^* \\
\textit{global} & ::= & \textit{type}_{\textit{val}} \; (\textit{import} \mid \textit{init}) \; \textit{export}^* & \textit{init} & ::= & \textit{instr}^* \\
\textit{start} & ::= & \textit{idx}_{\textit{func}} & \textit{type}_{\textit{val}} & ::= & \textbf{i32} \mid \textbf{i64} \mid \textbf{f32} \mid \textbf{f64} \\
\textit{table} & ::= & \textit{import}^? \; \textit{idx}_{\textit{func}}{}^* \; \textit{export}^* & \textit{type}_{\textit{func}} & ::= & \textit{type}_{\textit{val}}{}^* \rightarrow \textit{type}_{\textit{val}}{}^* \\
\textit{memory} & ::= & \textit{import}^? \; \texttt{byte}^* \; \textit{export}^* & \textit{idx}_{\textit{func}\,|\,\textit{global}\,|\,\textit{local}} & \in & \mathbb{N} \\
\textit{instr} & ::= & \multicolumn{4}{l}{\textit{type}_{\textit{val}}\textbf{.const} \; \textit{value} \mid \textit{type}_{\textit{val}}\textbf{.load} \mid \textit{type}_{\textit{val}}\textbf{.store} \mid \textbf{memory.grow}} \\
 & \mid & \multicolumn{4}{l}{\textbf{call} \; \textit{idx}_{\textit{func}} \mid \textbf{call\_indirect} \; \textit{type}_{\textit{func}} \mid \textbf{return} \mid \cdots}
\end{array}$$

Fig. 1. Excerpt from the abstract syntax of a simplified form of Wasm [32].

We show in experiments that nearly all benchmarks produced with Wasm-R3 spend the majority of their execution time in the original functions, not in replay code.

Our evaluation applies Wasm-R3 to real-world web-based Wasm applications, demonstrating that the approach is effective in creating 27 realistic and standalone benchmarks. We show that our optimizations effectively reduce the size of the recorded trace (by 99.53%, on average) and the size of the replay benchmark (by 9.98%, on average). The generated replay code accounts only for 0.20% of the total execution time, and hence, the extracted benchmarks accurately represent the original application. We release the benchmark suite created by Wasm-R3 during our evaluation, called *Wasm-R3-Bench*, to the community, and envision them to serve as a new standard for realistic and standalone Wasm benchmarks.

In summary, this paper contributes the following:

- We introduce the first record and replay technique for Wasm. It does not require support from or modification of the Wasm host environment, hardware, operating system, language runtime, or source compiler.
- We demonstrate the technique via a system which records execution traces of web applications in any browser and produces replay benchmarks that execute without any host environment—on any engine.
- We present optimization techniques that reduce trace size and improve replay performance. For several applications, these optimizations are vital to making our approach feasible at all, avoiding out-of-memory errors and excessive slowdowns that make benchmarks unrepresentative of the original application's performance.
- We demonstrate that Wasm-R3 is effective in real-world scenarios by using the approach to create benchmarks from 27 real-world web applications.
- We make Wasm-R3, associated tools, and the created benchmarks available as open source https://github.com/sola-st/wasm-r3.

2 Background

In this section, we provide necessary information to describe Wasm-R3. Figure 1 shows an excerpt of a simplified abstract syntax of Wasm. A Wasm *module* denotes a single binary file and consists of functions, global variables, an optional start function, and one table and memory. A *function* takes parameters, declares local variables, executes body instructions, and returns a sequence of results. A *global variable* stores a single value and can be accessed from all functions and can be either mutable or immutable. A *start function* is automatically executed when the module is loaded. A *table* maps function indices to opaque references to either extern (host) objects or Wasm functions. Tables can be used for indirect function calls via the **call_indirect** instruction. A *memory* represents a contiguous, byte-addressable, page-sized mutable array of memory. All of these entities can either be imported from a host execution environment, specifying a module and name pair, or exported

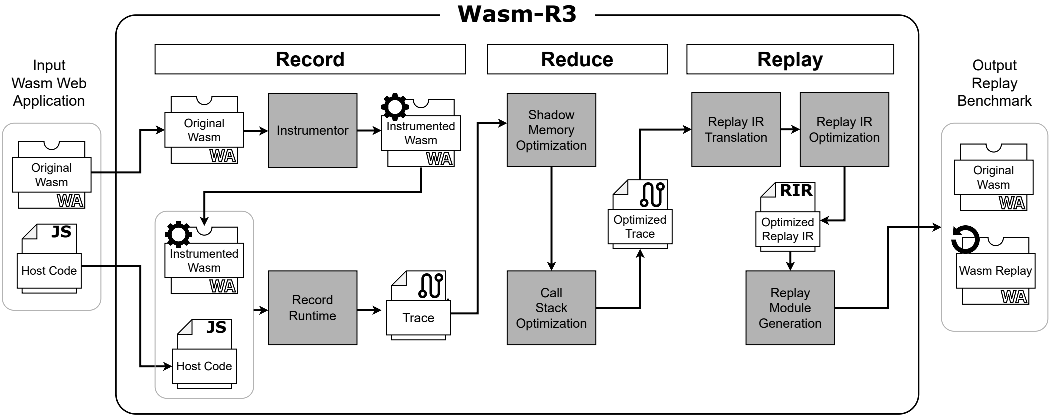


Fig. 2. Overview of the Wasm-R3 main phases and components.

under one or more names, allowing them to be accessed externally. Apart from these entities, modules can include initialization data for tables and memories.

One concept that plays a key role in Wasm-R3 is embedding of Wasm modules into a host environment. Host environments load Wasm modules, resolve imports and exports between modules, and provide host functions as imports to Wasm functions. Host functions can access state outside modules and perform I/O. While the web was the main motivation for Wasm initially, it was designed to be embedded in multiple environments [27]. Thus, although the web and JavaScript embedding was the primary one at launch, WASI [14] has emerged as a set of standard system APIs in non-web use cases. In principle, unlimited embedders are possible due to the environment-agnostic design of Wasm. After an embedder loads, verifies, and processes Wasm code, it provides bindings to a module's imports and creates the module's storage. The result is an *instance*, a runtime representation that contains the state of the module. A Wasm instance interacts with host environments by calling imported host functions and being called by exported Wasm functions.

Consideration of host functions introduces another important aspect of Wasm for Wasm-R3: nondeterminism. Since its inception, one of Wasm's explicit goals [27] has been to provide deterministic semantics across different hardware. However, there are three exceptions: NaN payloads, resource exhaustion, and host functions. Some Wasm instructions output non-deterministic NaN bit patterns in the presence of non-canonical input NaNs, as hardware may behave differently and canonicalizing all NaNs is deemed too expensive. Resources like memory obviously vary from host to host and computer to computer, so deep recursion or **memory.grow** might fail at different points, and of course a host function can perform I/O or even arbitrary updates to a Wasm instance's exported state. In this work, we focus on nondeterminism arising from the interaction with host functions.

3 Approach

3.1 Overview

Figure 2 gives an overview of Wasm-R3. Given a Wasm-based web application, Wasm-R3 executes the application, possibly with user input, and produces a benchmark that replays that execution without any user input. Wasm-R3 consists of three phases: *record*, *reduce*, and *replay*.

The record phase can be considered the frontend of Wasm-R3. It takes a Wasm web application as an input and produces a *trace* as a result. We assume that the application consists of Wasm modules and host code, e.g., in JavaScript. As the first step, Wasm-R3 intercepts the Wasm modules before they are loaded into the application and instruments the modules with recording logic. Our instrumentation tracks all function calls and returns as well as all loads and stores to mutable state. Then, the instrumented Wasm web application is run within the *record runtime*. While the user interacts with the web application, each instrumented Wasm module records its own execution trace. Note that if the input Wasm web application loads multiple Wasm modules, then the record phase produces one trace for each module.

For complex applications, the traces can grow prohibitively large. Thus, for efficiency, the reduce phase filters out unnecessary events from the output trace of the record phase. Specifically, it applies the *shadow memory optimization* and *call stack optimization* on the trace. By applying these optimizations, we only keep the parts of traces that are directly related to the nondeterminism that occured during execution. Eventually, the reduce phase yields optimized traces as outputs.

The replay phase is the backend of Wasm-R3. It takes the original, uninstrumented Wasm module and the corresponding optimized trace as inputs and produces a self-contained, executable *replay benchmark*. The replay phase does not modify the original module's functions. Instead, it simply merges them with generated *replay functions* to complete the executable replay benchmark. It first translates the input trace to an intermediate representation, the *replay IR*. Then, it applies *replay IR optimizations* to reduce the size of the IR and ultimately the generated code, ensuring that the resulting binaries satisfy the size restrictions commonly imposed by engines. Finally, it generates the replay benchmark from the optimized replay IR. As we discuss in Section 3.4.4, our replay binary generator supports three different output formats. This approach allows replay benchmarks to be executed in diverse environments, including web browsers and standalone Wasm runtimes.

The next sections dive into the details of the record phase (Section 3.2), trace reduction techniques (Section 3.3), and the replay phase (Section 3.4).

3.2 Record Phase

The record phase is responsible for recording the necessary information into a trace for reconstructing the benchmark that deterministically replays the Wasm web application's behavior. As described earlier, non-determinism gets introduced to Wasm applications by functions imported from the host environment. The record phase thus needs to capture information about function calls across the boundary between the host environment and the target Wasm module. This includes side effects to the Wasm module's mutable state, e.g., the memory section of the Wasm module, caused by host functions.

The following describes the format of traces (Section 3.2.1) and how these traces are recorded via instrumentation (Section 3.2.2).

3.2.1 Trace Structure. We define a *trace* data format that stores all necessary information about host function execution and their side effects. By defining traces, we effectively decouple the record phase and replay phase and relay any required data by the fixed format of traces.

Listing 3 provides type definitions of the trace structure. A `Trace` is a linear sequence of events. `Events` correspond to units of behavior that happen during the Wasm app execution and that possibly involve interaction with host code. There are six types of possible trace events: `FuncEntry`, `FuncReturn`, `Call`, `CallReturn`, `Load`, and `Store`. The `funcidx` field in events corresponds to the function index of the Wasm function. `ValType` corresponds to Wasm's four primitive types. A `FuncEntry` event corresponds to the start of the function body. This event represents the entrance to a Wasm-side function, with `params` storing the arguments given to the function call. Conversely, a `FuncReturn` event corresponds to the end of the function body. This event represents the exit from a Wasm-side function, with `values` storing the return values of the function. `Call` and `CallReturn`

```
Trace      = Seq<Event>
Event      = FuncEntry | FuncReturn | Call | CallReturn | Load | Store
FuncEntry  = { funcidx: I32, params: Seq<ValType> }
FuncReturn = { funcidx: I32, values: Seq<ValType> }
Call       = { funcidx: I32 }
CallReturn = { funcidx: I32, results: Seq<ValType> }
Load       = { memidx: I32,
               address: I32,
               value: ValType | I8 | I16 }
Store      = { memidx: I32,
               address: I32,
               value: ValType | I8 | I16 }
ValType    = I32 | I64 | F32 | F64
```

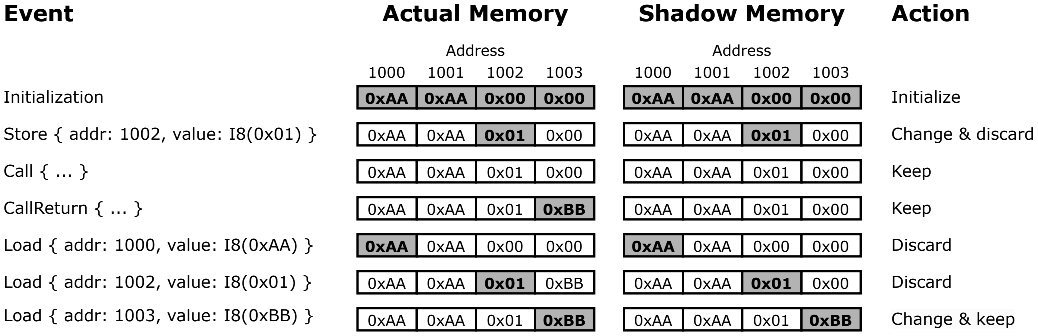
Fig. 3. Type definitions of the trace structure.

events are produced by calling and returning from functions imported by a Wasm module. Both events store the function index of the callee function, with the `CallReturn` event additionally storing the returned values. A `Load` event corresponds to Wasm **load** instructions. It stores the memory index in the `memidx` field, the address in the `address` field, and the loaded value in the `value` field. Similarly, a `Store` event stores the memory index, address and stored value of Wasm **store** instructions.

In practice, we encode the trace in two formats: a textual format and a binary format. The textual format is a JSON-like representation of the trace, where each trace event is encoded as an object, and the whole trace is encoded as a list of such objects. For instance, a `Load` event may be represented as `Load { memidx: I32(0), address: I32(1000), value: I16(300) }`. We use the textual format for human-reading purposes, such as debugging. In the binary format, each entry starts with a byte indicating the entry type, so the byte length of the entry is known. We use the binary format in our implementation to reduce memory usage and for efficient parsing of traces.

3.2.2 *Instrumentation.* We use an instrumentation-based approach to record interactions between the Wasm module and the host environment and to store them into traces. To this end, we add instructions to the Wasm binaries to capture runtime information, such as operand stack values. Using instrumentation allows us to record traces from arbitrary Wasm-based web applications, without changing the web browser implementation or depending on features specific to certain platforms or libraries.

An important property of our instrumentation is that it should not change the original Wasm module's semantics. Instead, the instrumented Wasm module serves as a drop-in replacement for the original Wasm module during recording, with the only behavioral change being the recording of a trace. To preserve the Wasm module's semantics during instrumentation, our instrumentation strategy uses special recorder functions, which take runtime information as input parameters, records the event into the trace, and return. We define recorder functions for each trace event. For instance, the recorder function for a `load` instruction gets `memidx`, `address`, `value` as input arguments and records the corresponding `Load` event. Our instrumentation copies the runtime information on the stack, calls the imported recorder function, and then returns to the original execution flow. By Wasm's function call semantics, calling the recorder function will consume only the copied values from the stack and will not divert the original control flow.



Event	Actual Memory				Shadow Memory				Action
	Address				Address				
	1000	1001	1002	1003	1000	1001	1002	1003	
Initialization	**0xAA**	**0xAA**	**0x00**	**0x00**	**0xAA**	**0xAA**	**0x00**	**0x00**	Initialize
Store { addr: 1002, value: I8(0x01) }	0xAA	0xAA	**0x01**	0x00	0xAA	0xAA	**0x01**	0x00	Change & discard
Call { ... }	0xAA	0xAA	0x01	0x00	0xAA	0xAA	0x01	0x00	Keep
CallReturn { ... }	0xAA	0xAA	0x01	**0xBB**	0xAA	0xAA	0x01	0x00	Keep
Load { addr: 1000, value: I8(0xAA) }	**0xAA**	0xAA	0x00	0x00	**0xAA**	0xAA	0x00	0x00	Discard
Load { addr: 1002, value: I8(0x01) }	0xAA	0xAA	**0x01**	0xBB	0xAA	0xAA	**0x01**	0x00	Discard
Load { addr: 1003, value: I8(0xBB) }	0xAA	0xAA	0x01	**0xBB**	0xAA	0xAA	0x01	**0xBB**	Change & keep

Fig. 4. Example of the shadow memory optimization.

3.3 Reduce Phase

The execution of a Wasm web application can produce millions of host interaction events. A naïve approach would quickly run out of time and memory before a replay binary is generated. An essential component to make Wasm-R3 practical is to reduce the size of traces by filtering out events that contain redundant or unnecessary information. We call this process *trace reduction*. A key insight is that we only need to keep trace events related to non-determinism. For instance, many of the `Store` events can be deterministically replayed by the original Wasm module itself. Thus, we can conclude that `Store` events are not necessary to replay host-side non-determinism.

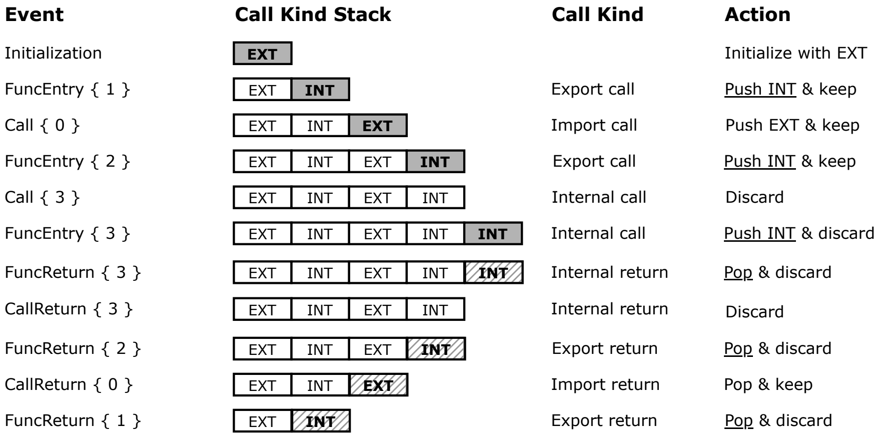
In this section, we describe two trace reduction techniques: *shadow memory optimization* (Section 3.3.1), which filters out redundant `Store` and `Load` events, and *call stack optimization* (Section 3.3.2), which filters out unnecessary `Call` and `CallReturn` events.

3.3.1 Shadow Memory Optimization.

Most of the `Store` and `Load` events in traces are not related to non-determinism and can thus be removed. It is not necessary to record all Wasm stores for accurate replay, as the original code performs the exact same sequence of stores as long as the current state of the program is the same. Thus, it is only necessary to keep stores related to non-deterministic behavior, i.e., stores that come from the host. Unfortunately, we cannot directly hook into host-side stores (e.g., when JavaScript writes into WebAssembly memory), since we only instrument the application's Wasm code. However, `Load` events on the Wasm side can observe when values in linear memory diverge from what was last recorded, which means they were modified by the host.

Inspired by memory optimization techniques in prior work [43, 50], we apply the *shadow optimization technique* to remove unnecessary `Store` and `Load` events and keep only `Load` events that observes the host-side side effect. The technique maintains a data structure called *shadow memory*, which keeps track of the written values to the original Wasm module's linear memory. By comparing the loaded value of a `Load` event and the value stored in the shadow memory, we can determine if the `Load` event observes the host-side side effect or not, and discard the unnecessary events.

Figure 4 illustrates how the shadow memory optimization works. For each step of a Wasm module's execution, we illustrate the corresponding trace event, the state of the module's actual memory, and the state of the shadow memory. For presentation brevity, we omit the irrelevant fields in the trace events. In the actual memory states, we represent the parts that are read or written as gray cells. In the shadow memory states, we represent the parts that are modified or compared with the loaded values as gray cells.



Event	Call Kind Stack	Call Kind	Action
Initialization	**EXT**		Initialize with EXT
FuncEntry { 1 }	EXT **INT**	Export call	Push INT & keep
Call { 0 }	EXT INT **EXT**	Import call	Push EXT & keep
FuncEntry { 2 }	EXT INT EXT **INT**	Export call	Push INT & keep
Call { 3 }	EXT INT EXT INT	Internal call	Discard
FuncEntry { 3 }	EXT INT EXT INT **INT**	Internal call	Push INT & discard
FuncReturn { 3 }	EXT INT EXT INT ▨INT	Internal return	Pop & discard
CallReturn { 3 }	EXT INT EXT INT	Internal return	Discard
FuncReturn { 2 }	EXT INT EXT ▨INT	Export return	Pop & discard
CallReturn { 0 }	EXT INT ▨EXT	Import return	Pop & keep
FuncReturn { 1 }	EXT ▨INT	Export return	Pop & discard

Fig. 5. Example of the call stack optimization.

We first initialize the shadow memory to contain the same values as the original Wasm module, by following the Wasm module's data section definition. The first trace event is a `Store` of a single-byte value `0x01` to address `1002`. The shadow memory optimization applies the same write to the shadow memory and discards the event. Then, as a result of the call of an imported function in the second and third events, the content at address `1003` is mutated. The fourth event is a `Load` of value `0xAA` from address `1000`. The optimization first compares the loaded value with the value at the same address in the shadow memory. Because the values are equal, the optimization discards the `Load` event. Similarly, the optimization discards the fifth event. The sixth event is a `Load` of value `0xBB` from address `1003`. The optimization compares the loaded value and the shadow memory value. Here, the values are different, i.e., the value has been mutated as a side effect of interacting with the host. Our optimization updates the shadow memory value as the loaded value and keeps the `Load` event. As a result of shadow memory optimization, only the second, third, and sixth trace events are remaining, whereas the other, unnecessary `Store` and `Load` events are discarded.

3.3.2 Call Stack Optimization. While we record every trace event related to function execution, a significant portion of them are unrelated to non-determinism. During the execution of a Wasm module, there are three possible kinds of function calls: *export calls*, *import calls*, and *internal calls*. Export calls are function calls from the host-side code to functions exported by the Wasm module. Import calls are function calls from the Wasm module to functions imported from the host environment. Internal calls are function calls from a function in the Wasm module to another internal function. Among these three kinds of calls, we do not need to keep track of internal calls as those can be deterministically replayed by the original Wasm code. Thus, we can safely remove `FuncEntry`, `FuncReturn`, `Call`, and `CallReturn` events produced by internal calls. In addition, we can remove `FuncReturn` events produced by returning from export calls. This is because the same return values can be deterministically replayed by the functions defined in the original Wasm module.

We apply *call stack optimization* to remove trace events produced by internal calls. To distinguish if an event was produced by an internal call or not, we track function calls in our own *call kind stack*. The call kind stack stores `INT` and `EXT` objects, which correspond to Wasm-internal contexts and host-side code contexts, respectively. Similar to conventional function call stacks, our call kind stack keeps track of calling contexts of trace events. By observing the stack top element and the next trace event, we can determine if the event was produced by an internal call and discard it.

Figure 5 illustrates our call stack optimization process. In this example, we assume that the function at index `0` is the only function imported from the host environment, and any other functions are internal functions of the original Wasm module. We represent trace events by omitting unnecesary fields and only annotating the kind of an event and the `funcidx` field as a number. The call kind stack column represents the state of the call kind stack at each step. Gray-colored cells represent objects pushed in the current step, and gray-hatched cells represent objects popped in the current step. The call kind column represents the kind of the function call produced by the corresponding trace event and whether the function is being called or returning. The rightmost column shows which action the call stack optimization performs at each step. The underlined actions are always performed regardless of the event's `funcidx` or the call kind stack top content. Note that we omitted `Store` or `Load` events from the trace for simplicity.

The call stack optimization rules are as follows. We first initialize the call kind stack with a single `EXT` object. This represents the outermost host execution context, e.g., JavaScript code which might call into exported Wasm functions. Then, we iterate over trace events. On a `FuncEntry` event, we first observe the stack top. If the top is `EXT`, then the event was produced by an export call; we keep the event. If the top is `INT`, then the event was produced by an internal call; we discard the event. Finally, we always push `INT` on the stack. On a `Call` event, we first check the `funcidx` field. If the index corresponds to an imported function, then the event was produced by an import call; we push `EXT` on the stack and keep the event. If the index corresponds to an internal function, then the event was produced by an internal call; we discard the event. On `FuncReturn` event, we first pop an object from call kind stack and discard the event. On a `CallReturn` event, we first check the `funcidx` field. If the index corresponds to an imported function, then the event was produced by returning from an import call; we pop an object from the stack and keep the event. If the index corresponds to an internal function, then the event was produced by returning from an internal call; we discard the event.

As illustrated in Figure 5, the call stack optimization discards all events produced by internal calls. In addition, it discards all `FuncReturn` events produced by returning from export calls. In the example, this filters out 6 out of 10 events. The effect on real-world traces is evaluated in Section 5.

3.4 Replay Phase

In the replay phase, Wasm-R3 gets an input trace and generates an excutable, standalone replay benchmark. In this process, Wasm-R3 uses a *replay intermediate representation*, or *replay IR*. We describe the definition of the replay IR in Section 3.4.1. By first generating a replay IR (Section 3.4.2) from a trace, we can apply the replay IR optimizations (Section 3.4.3) to reduce the size of the replay IR. Finally, Wasm-R3 translates this replay IR into one of three different output formats (Section 3.4.4) and generates a replay binary (Section 3.4.5).

3.4.1 Replay IR. Replay IR is a format designed to represent behaviors of *replay functions*. Replay functions are functions that implement the replay mechanism by replaying the return values and side-effects of host-side functions recorded during the record phase. The goal of the replay phase is to generate a replay function for each function imported from the original Wasm module. We utilize the replay IR as a general format that describes the behaviors of replay functions without depending on specific output formats. Thus, introducing the replay IR effectively divides the problem of generating replay binaries for multiple output formats into three problems: 1) translating a trace into a replay IR, 2) optimizing the replay IR, and 3) translating the replay IR to each output format.

We present the definition of the replay IR in Figure 6. An `Action` corresponds to a single instruction in the host: `ExportCall` or `MutateMem`. An `ExportCall` represents a function call from the host to an exported Wasm function. A `MutateMem` represents a host-side effect of mutating the Wasm module's

```
Action      := ExportCall | MutateMem
ExportCall  := { idx: I32, vals: Seq<ValType> }
MutateMem   := { idx: I32, addr: I32, val: I8 }
ValType     := I32 | I64 | f32 | f64
Context     := Seq<Action>
Function    := Seq<Context>
Replay      := Map<I32, Function>
```

Fig. 6. Type definitions of the replay IR.

```
function translate(optimized_trace: Trace) -> (Replay, Function):
  let GLOBAL_CONTEXT: Context = new Context()
  let ENTRY_FUNC: Function = new Seq(GLOBAL_CONTEXT)
  let last_context: Ref<Context> = &GLOBAL_CONTEXT
  let context_stack: Stack<Ref<Context>> = Stack().push(&GLOBAL_CONTEXT)
  let replay: Replay = Map()
  foreach event in optimized_trace:
    switch event:
      case FuncEntry:
        context_stack.top().push(new ExportCall(event))
      case Call:
        let new_context = new Context()
        replay.get(event.funcidx).append(&new_context)
        context_stack.push(&new_context)
        last_context = &new_context
      case CallReturn:
        last_context = context_stack.pop()
      case Load:
        let new_action = new MutateMem(event)
        if typeof last_context.last() == ExportCall:
          last_context.splice(last_context.length - 1, new_action)
        else:
          last_context.append(new_action)

  return (replay, ENTRY_FUNC)
```

Fig. 7. Trace to replay IR translation algorithm.

linear memory content. A `Context` is a sequence of actions, which represents the actions executed during the context of a single function call. Then, we define a `Function`, which is a sequence of contexts. Each `Context` corresponds to the instructions executed by i-th invocation of a host-side function, where i is the index of the `Context` in the `Function` sequence. Finally, we define `Replay` as a mapping from `I32` numbers to `Function`s; the `I32` numbers represent the function indices of the functions imported by the original Wasm module, and the mapped `Function`s represent the corresponding replay functions.

3.4.2 From Trace to Replay IR. We present the algorithm to translate a trace into a replay IR in Figure 7. The algorithm gets an input trace and returns a `Replay` and a `Function`. The `Replay` value corresponds to the map from the imported function indices to the replay functions. The `Function` value corresponds to the entry function to the Wasm module; as it does not correspond to a function index of the original Wasm module, we return it separately from the `Replay` value.

We use five variables in the algorithm: `GLOBAL_CONTEXT`, `ENTRY_FUNC`, `last_context`, `context_stack`, and `replay`. The variable `ENTRY_FUNC` denotes the entry `Function`. Because the entry function is executed exactly once, it has only one `Context`, which is the variable `GLOBAL_CONTEXT`. The variable `last_context` is a reference to a context; it points to the latest context that new `MutateMem` actions should be appended to. The variable `context_stack` is a stack of references to contexts. We maintain `context_stack` to keep track of the current host-side function call and push actions into correct contexts. We initialize `last_context` and `context_stack` with `GLOBAL_CONTEXT` as the entry function is the first function executed in the whole Wasm web application execution. Finally, the variable `replay` stores the `Replay` object which will be returned. We assume that each `Function` object inside the `replay` is properly initialized with an empty sequence when we access it.

The goal of the algorithm is to fill the `Functions` in the `replay` with `Contexts` and fill the `Contexts`, including `GLOBAL_CONTEXT`, with `Actions`, so that the return values correctly mirror the host-side behaviors recorded in the input trace. To correctly mirror the input trace, 1) a correct `Action` must be created for each event and 2) the created `Action` must be inserted to a correct `Context`. To do this, the algorithm uses case analysis on each event and properly updates `last_context` and `context_stack`. For a `FuncEntry` event, it creates an `ExportCall` action and pushes it to the `Context` of the current function. We find the `Context` of the current function by referring to the top of `context_stack` in this case. For a `Call` event, it creates a new `Context` object, namely `new_context`, for this import call. This new `Context` object is appended to the `Function` at the `event.funcidx` index of `replay`. Then, it pushes the reference to `new_context` onto `context_stack` and updates `last_context`. For a `CallReturn` event, the current function call is returned, so it pops from `context_stack`. Then, by updating `last_context` to the popped `Context` object, when we append `MutateMem` objects to `last_context`, it correctly mirrors the fact that the side effect observed by the Wasm-side execution is caused by the most recently returned host-side execution. Lastly, for a `Load` event, it creates a new `MutateMem` action reflecting the side effect on the memory. Then, to insert this action into the correct position, it inspects the last element of `last_context`. In most of the cases, it appends the new `MutateMem` at the last position of `last_context`. However, if the last action on the `last_context` is an `ExportCall` action, this means that the mutation happened before the corresponding export call and the side effect was later observed. Thus, it inserts the new `MutateMem` action right before the `ExportCall` action.

3.4.3 Replay IR Optimizations. We apply optimizations on replay IRs to reduce their size. Before we generate an output replay benchmark, we reduce the replay IR size so that the size of the replay binary is also reduced. By reducing the size of the replay binary, we address three practical issues.

- First, we produce valid replay binaries that are accepted by web browser engines. Production web engines impose a common restriction on the size of functions inside the Wasm binary[1]. By naïvely translating `Function` objects in a replay IR into Wasm functions, the size of replay functions may exceed the function size limit. To solve this issue, we employ replay IR optimizations to reduce the number of instructions of each replay function and produce smaller Wasm replay functions.
- Second, we shorten the compilation time of the replay binaries, which correlates with the runtime performance of the replay benchmark. A previous study [52] has found out that the compilation time of Wasm modules significantly impacts the performance evaluation of a Wasm engine. Since our replay IR optimization techniques reduce the overall size of the replay binary, we can reduce the impact of compilation time on the performance evaluation of Wasm engines.

[1] See, e.g., https://github.com/v8/v8/blob/master/src/wasm/wasm-limits.h.

```
MutateMem {idx: 0, addr: 0, val: \08}
MutateMem {idx: 0, addr: 1, val: \07}
MutateMem {idx: 0, addr: 2, val: \06}
MutateMem {idx: 0, addr: 3, val: \05}
MutateMem {idx: 0, addr: 4, val: \04}
MutateMem {idx: 0, addr: 5, val: \03}
MutateMem {idx: 0, addr: 6, val: \02}
MutateMem {idx: 0, addr: 7, val: \01}
MutateMem {idx: 0, addr: 8, val: \00}
```

(a) Replay IR before the optimization.

```
BulkMutateMem {
  idx: 0,
  addr: 0,
  val: "\08\07\06\05\04\03\02\01\00"
}
```

(b) Replay IR after the optimization.

Fig. 8. Memory write merge optimization example.

- Third, we reduce the execution time of the replay benchmarks by reducing the number of instructions in the final replay function bodies. We achieve this effect by merging mulitple instructions into a single instruction while preserving the function behavior.

We describe our two replay IR optimization techniques: *memory write merge* and *function split.*

Memory Write Merge Optimization. By inspecting replay IRs, we found that Wasm often involves writing data to consecutive bytes in the Wasm linear memory. Inspired by the **memory.init** instruction, which executes bulk writes on multiple bytes, we design the *memory write merge* optimization that merges multiple `MutateMem` actions on consecutive addresses in a single action. To represent the bulk write, we introduce a new action `BulkMutateMem`. The `BulkMutateMem` action has the same syntax as `MutateMem`, except that the `val` field is a number type with no size limit. In theory, the memory write merge optimization can merge an unlimited number of `MutateMem` actions into a single `BulkMutateMem` action. We translate the `BulkMutateMem` action into the **memory.init** Wasm instruction.

Figure 8 illustrates an example of applying the memory write merge optimization. The left-side figure is a part of the replay IR before applying the optimization. It is a sequence of eight `MutateMem` actions on consecutive bytes. The optimization merges the eight seperate writes in eight `MutateMem` actions into a single, 8-byte value in the `BulkMutateMem` action.

Function Split Optimization. As mentioned earlier, production web browser engines impose a common restriction on the size of a single Wasm function. We observed that some `Function` objects in replay IR exceed that maximum size limit. These `Function` objects represent functions frequently called in the Wasm web application execution. For instance, we observed a utility function that performs a conversion of `float64` values to integers corresponding to millions of actions.

To prevent our output replay binaries from being invalid, we employ *function split optimizations* on such large `Function` objects. In the function split optimization, we outline some parts of a `Function` object into other `Function` objects and replace them with calls to the new `Function` objects. By setting an appropriate threshold of the `Function` object size, we maintain each `Function` object size below the maximum Wasm function size limit imposed by the web engines.

3.4.4 Output Formats. The replay benchmarks generated by Wasm-R3 in the replay phase consist of three parts. First, the original Wasm module of the application. Second, the *replay code*, which replicates the behavior of the host environment during the record phase. Third, the *setup and instantiation code*, which links the first two parts together. The setup and instantiation code fulfills the imports of the original Wasm module with functions from the replay code and starts the replay benchmark. The last two parts are generated from the replay IR.

While the original application code is always in Wasm, we can make different choices for the replay code and the setup and instantiation code. If the replay code is in JavaScript, the setup and instantiation code should also be in JavaScript (option “JS"). When the replay code is in Wasm, we can either statically link the replay code with the original Wasm module into a single, self-contained binary (option “self-contained Wasm"), or generate JavaScript setup and instantiation code that loads the replay code and the original Wams module separately (option “dynamic linking"). In the dynamic linking option, the actual linking between the original Wasm module and the replay code happens during the instantiation at runtime.

All three output formats are sensible choices. A self-contained Wasm replay is the easiest option for downstream consumers of the benchmark, as it can be executed in any Wasm engine, including Wasm standalone runtimes that do not have a JavaScript host environment. We also expect a self-contained Wasm is the most performant way for replay because it does not involve function calls across the Wasm module and JavaScript host environments. Hence, we choose the self-contained Wasm format by default and use it for evaluation in Section 5. However, when the replay code is kept in a separate Wasm module or in JavaScript, this can be useful to benchmark cross-language or multi-module interactions. In Wasm-R3, we provide options to select from three output formats, so users can generate replay benchmarks according to their use cases.

3.4.5 From Replay IR to Output Formats. Once we have a replay IR, the generation of an executable, standalone replay benchmark is carried out in a straightforward, single-pass manner. For each `Function` in the replay IR, we generate a replay function according to the output mode. For JavaScript, this would be a JavaScript function, and for Wasm, this would be a Wasm function. We then define a global counter variable for each function to keep track of the current `Context`. The body of the generated function consists of a switch statement in the respective language that maps different counter values to different sequences of instructions, followed by a part that increments the counter for each invocation. Each sequence of instructions that are translated from a `Context` is a series of simple line-by-line translations of `Actions` in the replay IR to their corresponding instructions in the langauge. For example, `MutateMems` are translated to their corresponding **store** instructions.

4 Implementation

In this section, we describe notable implementation details of Wasm-R3.

Implementation Summary. The implementation of Wasm-R3 amounts to roughly 2,200 lines of TypeScript and 2,200 lines of Rust, divided into the frontend (the record and reduce phases) and the backend (the replay phase). The code is released under the MIT License and is publicly available at https://github.com/sola-st/wasm-r3. Wasm-R3 utilizes two third-party libraries: Wasabi [32], an instrumentation framework for Wasm, which we use to inject calls to recorder functions and to store corresponding trace events, and Binaryen [21], a compiler toolchain and infrastructure, which we use to optimize replay binaries.

Proxy. Rather than intrusively modify web browsers (or Wasm engines in the web browsers), we employ a *proxy* to intercept Wasm and JavaScript code. Modern web browsers expose the capability to intercept and modify network requests and responses. For instance, Chromium provides the DevTools Protocol [23] for this purpose. The proxy component leverages this capability to intercept JavaScript files and patch[2] the function definitions of Wasm module instantiation APIs: `WebAssembly.Instance`, `WebAssembly.instantiate`, and `WebAssembly.instantiateStreaming`. The new instantiation functions intercept the Wasm binary before instantiation and inject instrumentation before forwarding them to the underlying Wasm engine. We utilize the Playwright

[2] Of course, monkey-patching JavaScript has robustness issues, and some modules can be missed.

library [40] to implement the proxy across all major web browsers. This proxy approach allows Wasm-R3 to instrument every Wasm module on-the-fly without modifying the browser implementation.

Mutable State in Wasm. In Wasm, there are three kinds of mutable states in instances: globals, tables, and memories. Globals are mutable or immutable, single-value storage that can be imported and exported from Wasm modules. Tables store opaque references to host objects and Wasm functions. Since the **call_indirect** instruction indirects through a table, mutable tables can be used to implement dynamic linking via indirect calls. Like Wasm linear memory, globals and tables can be modified by the host environment if exported from an instance. Thus, in order to replay all nondeterminism from the host, we also need to record mutation of globals and tables. Although not described in detail here, Wasm-R3 also records and replays global and table mutations by instrumenting **global.get** and **table.get** instructions, respectively. Similarly, it also employs shadow memory optimizations for globals and tables to distinguish mutations from the host environment and the module itself.

Simultaneous Record and Reduce. In our implementation of Wasm-R3, the reduce phase is partially overlapped with the recording phase; the reduce phase filters out most redundant trace events even before they are stored in a trace. This is done primarily by the shadow memory optimization and the call stack optimization algorithms, which are applied online in the recorder functions. For example, a **load** instruction simply reads both the shadow and real memories and suppresses generating a load event if the two values are the same, which implies that the program either writes the value or has already observed a host-written value. Similarly, the call stack optimization algorithm is included directly in the recorder function. Filtering events requires more checks, but is less expensive than generating events and then later filtering them out, which naturally saves space but also reduces the overall recording overhead.

5 Evaluation

We evaluate Wasm-R3 by addressing the following four research questions.

- **RQ1. Applicability**: To what extent does Wasm-R3 apply to real-world web applications and different Wasm engines?
- **RQ2. Performance**: How much overhead does Wasm-R3's record phase introduce? What are the performance characteristic of the replay benchmarks?
- **RQ3. Effectiveness of trace reduction**: To what extent do our trace reduction techniques reduce the size of the recorded traces?
- **RQ4. Effectiveness of replay optimization**: By how much do our replay optimization techniques reduce the size of replay binaries? How do the optimizations impact the performance characteristics of the replay benchmarks?

5.1 Experimental Setup

We collect URLs of real-world *Wasm web applications*, which we define to be interactive webpages that load at least one Wasm module, to serve as our evaluation targets. To find such applications, we use two websites as starting points: *Made with WebAssembly* [55], which is an open-source website that showcases projects created with Wasm, and *Awesome-Wasm* [20], an open-source repository that lists Wasm-related webpages. We gathered URLs for Wasm web applications from these webpages by first manually crawling them and their subpages and filter out inaccessible websites, e.g. 404 or that require authentication, non-interactive webpages, and pages where the relevant Wasm

APIs, e.g. `WebAssembly.Instance`, `WebAssembly.instantiate` or `WebAssembly.instantiateStreaming`, are not used in any script.

For repeatability, we use test scripts that automatically interact with each website using the Playwright library [40] by mimicking common use cases. Each script executes multiple user actions, such as clicking a button or typing text into an input form. For websites that use multiple Wasm modules, we write multiple test scripts, each focusing on a different module. These scripts run against live sites, which of course evolve over time and automation breaks. In fact, the difficulties of reliably and reproducibly automating the execution of Wasm web applications is a key motivation of Wasm-R3, which ultimately produces completely self-contained Wasm benchmarks.

While writing test automation scripts, we exclude some applications from our evaluation targets that have one or more of the following problems:

(1) Throw errors even when used without Wasm-R3.
(2) Require external files or privileged information. For example, we remove a GameBoy emulator [8] because it requires a GameBoy ROM file that we could not supply.
(3) Take unreasonably long time to download required data from the network. This affects several video game ports, such as Arxwasm [15] and D3wasm [16].
(4) Though statically appear to use Wasm, don't dynamically load any Wasm modules. For example, we exclude Wasmboy [19] because we could not automate it to load Wasm.
(5) Require automation scripts that would interact with the HTML canvas element. As the Playwright library does not provide APIs to recognize images inside a canvas element, we cannot perform any meaningful interactions with such applications beyond randomly clicking inside the canvas.
(6) Exhibit flakiness in the automation scripts without any meaningful errors. For example, we exclude the application a tic-tac-toe game [37], because it sometimes fails to load the game.

As a result, we compiled 43 URLs to Wasm web applications. Table 2 summarizes our evaluation targets. Our evaluation targets are composed of 9 programming language applications, 8 Wasm benchmarks, 6 video games, 4 graphics applications, 3 media applications, 2 mathematical computation applications, 2 simulator applications, and 1 ML(machine learning) application. We claim these represent real-world Wasm web applications as we gathered them from well-known, open-source compilations of Wasm web applications and include applications from various domains.

We evaluate the benchmarks created with Wasm-R3 on three web browser engines (SpiderMonkey [3] version 125.0b7, V8 [4] version 12.5.149, and JavaScriptCore [2] version 277039) and three standalone Wasm engines (Wizard [51] version 24α.1998, Wasmtime [6] version 19.0.1, and Wasmer [5] version 4.2). We use the standalone-Wasm output format of the benchmarks for the entire evaluation (Section 3.4.4), as they can be executed across both web browser engines and standalone Wasm engines. To run a standalone-Wasm benchmark in a browser, we use a simple JavaScript wrapper that loads the replay benchmark and calls its entry function.

Our experiments are conducted on a machine running Ubuntu 22.04.1, equipped with an Intel Core i9-13900k CPU and 192GB of DRAM. With the Intel Core i9-13900k, we disable E-cores and use only P-cores, and set the Linux CPU frequency governor to performance mode for consistent results. We use Chromium 123.0.6312.4 as the browser for the proxy component in the record phase. For the experiments in RQ2 and RQ4, we repeat each measurement ten times for each target.

5.2 RQ1. Applicability

We evaluate the applicability of Wasm-R3 in two ways. First, we evaluate its ability to produce accurate benchmarks from various real-world Wasm web applications, which we call the *accuracy*

Table 2. List of evaluation target Wasm web applications.

Name	URL	Domain	Success
boa	https://boajs.dev/boa/playground	Progr. lang.	✓
bullet	https://magnum.graphics/showcase/bullet	Simulator	✓
commanderkeen	https://www.jamesfmackenzie.com/chocolatekeen	Video game	✓
factorial	https://www.hellorust.com/demos/factorial/index.html	Mathematics	✓
ffmpeg	https://w3reality.github.io/async-thread-worker/examples/wasm-ffmpeg/index.html	Media	✓
fib	https://takahirox.github.io/WebAssembly-benchmark/tests/fib.html	Benchmark	✓
figma-startpage	https://www.figma.com	Graphics	✓
fractals	https://raw-wasm.pages.dev	Graphics	
funky-kart	https://www.funkykarts.rocks/demo.html	Video game	✓
game-of-life	https://playgameoflife.com	Video game	✓
gotemplate	https://gotemplate.io	Progr. lang.	
guiicons	https://raylibtech.itch.io/rguiicons	Utility	✓
hnset-bench	https://raw.githack.com/gorhill/uBlock/master/docs/tests/hnset-benchmark.html	Benchmark	
hydro	https://cselab.github.io/aphros/wasm/hydro.html	Simulator	✓
image-convolute	https://takahirox.github.io/WebAssembly-benchmark/tests/imageConvolute.html	Benchmark	
jqkungfu	http://jqkungfu.com	Progr. lang.	✓
jsc	https://mbbill.github.io/JSC.js/demo/index.html	Progr. lang.	✓
lichess	https://lichess.org/analysis	Video game	
livesplit	https://one.livesplit.org	Utility	
mandelbrot	http://whealy.com/Rust/mandelbrot.html	Graphics	✓
multiplyDouble	https://takahirox.github.io/WebAssembly-benchmark/tests/multiplyDouble.html	Benchmark	✓
multiplyInt	https://takahirox.github.io/WebAssembly-benchmark/tests/multiplyInt.html	Benchmark	✓
ogv	https://brionv.com/misc/ogv.js/demo	Media	
onnxjs	https://microsoft.github.io/onnxjs-demo/#	ML	
pacalc	http://whealy.com/acoustics/PA_Calculator/index.html	Mathematics	✓
parquet	https://google.github.io/filament/webgl/parquet.html	Graphics	✓
pathfinding	https://jacobdeichert.github.io/wasm-astar	Benchmark	✓
playnox	https://playnox.xyz	Video game	
rfxgen	https://raylibtech.itch.io/rfxgen	Utility	✓
rguilayout	https://raylibtech.itch.io/rguilayout	Utility	✓
rguistyler	https://raylibtech.itch.io/rguistyler	Utility	✓
riconpacker	https://raylibtech.itch.io/riconpacker	Utility	✓
roslyn	http://roslynquoter-wasm.platform.uno	Progr. lang.	
rtexpacker	https://raylibtech.itch.io/rtexpacker	Utility	✓
rtexviewer	https://raylibtech.itch.io/rtexviewer	Utility	✓
rustpython	https://rustpython.github.io/demo	Progr. lang.	
sandspiel	https://sandspiel.club	Video game	✓
sqlgui	http://kripken.github.io/sql.js/examples/GUI	Progr. lang.	✓
sqlpractice	https://www.sql-practice.com	Progr. lang.	
takahirox	https://takahirox.github.io/WebAssembly-benchmark	Benchmark	
timestretch	https://superpowered.com/js-wasm-sdk/example_timestretching	Media	
waforth	https://el-tramo.be/waforth	Progr. lang.	
wheel	https://boyan.io/wasm-wheel	Benchmark	

experiment (Section 5.2.1). Second, we evaluate to what extent the produced benchmarks execute successfully across different Wasm engines, which we call the *portability experiment* (Section 5.2.2).

5.2.1 Accuracy Experiment. We evaluate how accurately Wasm-R3's replay benchmarks match their execution in Wasm web applications. The term "accurate" here means that the original web application and the corresponding replay benchmarks show the same behavior. We assess accuracy by recording traces of both executions and test if both traces are exactly the same.

Table 2 shows for each application whether we could successfully produce accurate replay benchmarks. In total, Wasm-R3 produces accurate replay benchmarks for 27 out of 43 applications. These applications cover a wide range of domains, including programming language applications, graphics applications, and video games. To the best of our knowledge, the resulting set of benchmarks is the first executable benchmark suite of real-world Wasm web applications. We refer to these benchmarks as *Wasm-R3-Bench* and, unless mentioned otherwise, use them throughout the rest of the evaluation.

Due to the complexity of the web and limitations of the libraries that Wasm-R3 uses, the approach fails to produce accurate benchmarks for the remaining 16 applications. We categorize the failures into three groups:

- *Implementation limitations (5 cases).* Some failures are due to the known limitations of our implementation. For IMAGE-CONVOLUTE, the trace contains over a million calls to a single host function. The current function split optimization works only inside a single context in the replay IR, which prevents the application of the optimization in this case; PLAYNOX fails for the same reason. We believe this would be resolved with a more advanced function split optimization that works across multiple contexts. For TIMESTRETCH, HNSET-BENCH, and WHEEL, our proxy logic does not seem to work the use cases of the applications.
- *Dependency limitations (4 cases).* Some failures are caused by the limitations of the libraries that Wasm-R3 uses. Wasabi fails to instrument FRACTALS and LICHESS, because they use the SIMD proposal, and LIVESPLIT, because it uses the threads proposal, which are not yet supported by Wasabi. Binaryen fails on WAFORTH because the library does not support block-type parameters.[3]
- *Unknown problems (7 cases).* For the remaining 7 applications, we could not determine the cause of the failure. We are currently investigating their cause.

5.2.2 Portability Experiment. The following experiment evaluates to what extent the replay benchmarks generated by Wasm-R3 execute successfully across different Wasm engines. We run the portability experiment with all 27 accurate replay benchmarks, trying to run them on three web browser engines and three Wasm standalone engines (Section 5.1). When running the portability experiment with web browser engines, we experiment with different optimization tiers of each engine. We count the experiment as successful if the replay benchmark runs successfully on all optimization tiers of the engine. Likewise, as the Wizard and Wasmer engines also provide different optimization tiers, we also experiment with them. All replay benchmarks successfully run across all execution tiers of the three web browser engines and three Wasm standalone engines. In summary, our experiments show that Wasm-R3 is applicable in various usage scenarios. In particular, we produce a suite of 27 replay benchmarks from real-world Wasm web applications, and these benchmarks run successfully on various Wasm engines.

5.3 RQ2. Performance

We evaluate the performance of Wasm-R3 from two perspectives. First, we assess the amount of overhead introduced during the recording phase (Section 5.3.1). Keeping the overhead low is crucial to minimize disruption to user interactions during the recording phase. Second, we examine the performance characteristics of the replay benchmarks by measuring the time spent in code of the original Wasm module and in code added by Wasm-R3 to enable replay (Section 5.3.2). For a replay binary to be useful for evaluating the performance of Wasm engines, the majority of time should be spent in the original Wasm code.

5.3.1 Record Overhead Experiment. The following experiment evaluates the overhead of Wasm-R3's record phase. We measure the total CPU cycles spent by the Chromium renderer process of the target application using the Linux perf tool. Among the 27 replay binaries in Wasm-R3-Bench, we exclude PARQUET in this experiment as our record overhead measurement infrastructure does not support its use of the WebGL library. To compute the recording overhead, we compare the performance of the application when running with an uninstrumented and an instrumented Wasm module. As in RQ1, we use our UI-level test scripts to simulate user interactions. Because the test scripts are at the UI level, there is a risk of having slightly different workloads in different runs.

[3] https://github.com/WebAssembly/binaryen/issues/6407

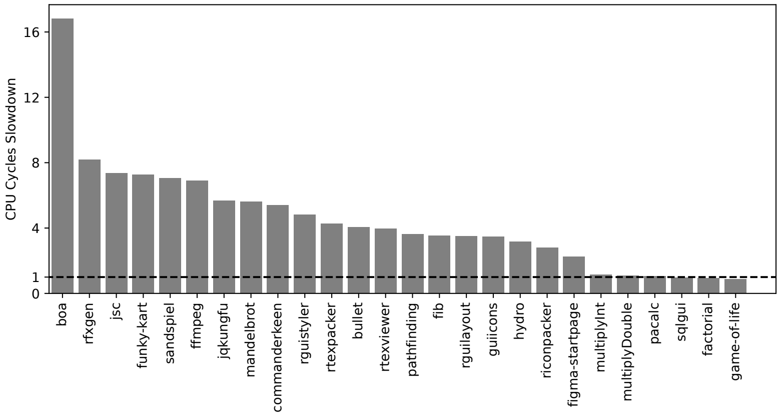


Fig. 9. Relative increase in CPU cycles spent by the Chromium renderer process over uninstrumented application (2.0 = twice as many cycles, 1.0 = same).

To mitigate this risk, and also to account for the inherent noise of performance measurements, we repeat each measurement across ten runs and compute the arithmetic mean of the spent CPU cycles.[4] Then, we compute the ratio of the arithmetic mean of the CPU cycles spent by the instrumented application to the arithmetic mean of the CPU cycles spent by the uninstrumented application.

Figure 9 shows the overhead introduced by the record phase of Wasm-R3. We measured the CPU cycles spent executing JavaScript and Wasm code with and without instrumentation. While the overhead varies across applications, the slowdown is generally modest, with a median of approximately 3.79×, a geometric mean of 3.40×, and all but one application exhibiting less than 8.18× overhead. In practice, this overhead is acceptable; applications are still interactive and users can record realistic usage scenarios with Wasm-R3.

5.3.2 *Replay Characteristics Experiment.* We now evaluate the performance characteristics of the replay benchmarks created by Wasm-R3. Wasm-R3 repackages the functions of the original Wasm module with replay functions, creating a standalone executable. While any Wasm workload can serve as a benchmark, the overall goal is to capture the performance characteristics of the *original* Wasm code. A key metric is then the proportion of the work spent in the original functions versus the replay functions. To answer this question, we first measure the CPU cycles per function using the `fprofile` monitor of Wizard. Then, we distinguish whether the function belongs to the original Wasm module or the replay functions. Summing the CPU cycles spent in each group, we can calculate the total cycles spent in the original Wasm module and the replay functions. We repeat the experiment ten times to get the arithmetic mean value of the CPU cycles to reduce variance in the measurements.

Figure 10 displays the results of this experiment. The upper portions of the bars in light gray color represent the percentage of the cycles spent in the functions from the original Wasm module, while the lower portions in red color represent the percentage of the cycles spent in replay functions. Ideally, a benchmark would spend 100% of the cycles in the original Wasm code and 0% in replay. For the benchmarks we gathered, we find the geometric mean of the cycles spent in the replay binary to be 0.20%. Half of the benchmarks spend less than 1.53%, all but one less than 24%, and the maximum

[4] For FIB, we compute the arithmetic mean of nine runs only, because one recording run failed due to flakiness.

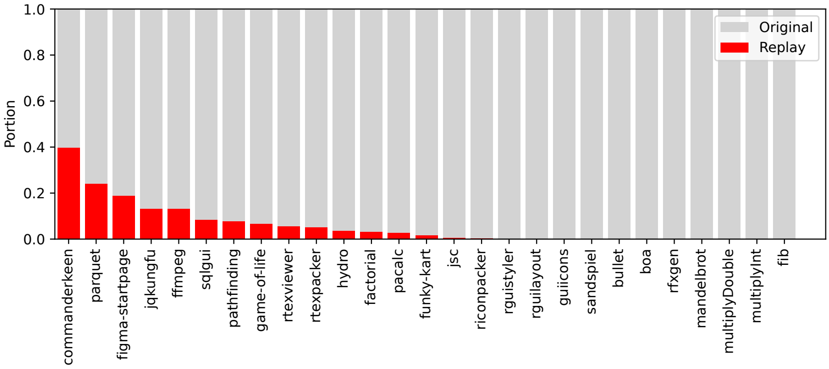


Fig. 10. Proportion of the execution spent in the original Wasm modules (light gray) and replay functions (red), in CPU cycles.

cycles spent in the replay part is 39%. The proportion varies according to the frequency and nature of the application's interactions with the host. Yet, recall that our technique only requires the replay return values and observed side-effects (e.g. memory modifications) from host calls. Side-effects from the host that are never observed by the application, as well as host functions that only read application memory, do not generate trace events. This leads to the somewhat counter-intuitive result that applications that *produce* a lot of data for the host (e.g. rendering frames for a game) actually do not generate many trace events; they generate (relatively small) trace events when the user interacts with the game.

Thus, in summary, our results show that the modules in Wasm-R3-Bench mostly exercise the behavior of the original, real-world applications, making these executions suitable for evaluating the performance of Wasm engines.

5.4 RQ3. Effectiveness of Trace Reduction

We evaluate how effective Wasm-R3's trace reduction techniques (Section 3.3) are in reducing the size of the recorded traces. We record traces with four variants of the approach: 1) without any trace reductions, 2) only with the shadow memory optimization, 3) only with the call stack optimization, and 4) with both optimization techniques. The fourth variant corresponds to the full Wasm-R3 approach. We target all 27 modules in Wasm-R3-Bench, measure the size of the traces produced by the four variants, and report the differences in them. By "the size of a trace," we here mean the number of trace events recorded by Wasm-R3.

Table 3 shows the results. Both trace reduction techniques are effective in reducing the trace size. On average, the shadow memory optimization and the call stack optimization reduce traces to 27.20% and 38.17% of the original trace size, respectively. Together, the two optimizations reduce the traces to only 0.47% of the original size, i.e. a more than 200× reduction. In the development process, before optimization, Wasm-R3 initially failed to produce traces for most applications due to the lack of memory or timeouts. However, after applying our trace reduction techniques, Wasm-R3 filters out a large portion of trace events and succeeds to produce traces for real-world applications. Hence, we believe that our trace reduction techniques are essential to produce replay benchmarks from real-world Wasm web applications.

From the table, three interesting cases emerge: multiplyDouble, multiplyInt, and fib. For multiplyDouble and multiplyInt, the optimizations do not remove any trace events. This is

Table 3. Trace reduction experiment results.

Name	Trace (# Events)			
	No-opt	Shadow-opt	Call-stack-opt	All-opt
boa	1422316	41.36%	58.65%	0.01%
bullet	60731762	14.27%	85.75%	0.01%
commanderkeen	190191109	21.12%	79.12%	0.23%
factorial	1483	35.87%	66.01%	1.89%
ffmpeg	83	36.14%	87.95%	24.10%
fib	4912039416	100.00%	0.00%	0.00%
figma-startpage	3065	61.01%	54.71%	15.73%
funky-kart	170064681	29.63%	71.86%	1.48%
game-of-life	2178	16.71%	85.31%	2.02%
guiicons	87593617	8.48%	91.53%	0.01%
hydro	4494	37.96%	62.35%	0.31%
jqkungfu	83	48.19%	80.72%	28.92%
jsc	1829171	69.12%	35.63%	4.75%
mandelbrot	151019238	0.90%	99.21%	0.11%
multiplyDouble	24	100.00%	100.00%	100.00%
multiplyInt	24	100.00%	100.00%	100.00%
pacalc	194039	43.05%	58.95%	2.00%
parquet	16736	54.92%	60.40%	15.32%
pathfinding	30798960	35.64%	64.44%	0.08%
rfxgen	143104877	6.79%	93.21%	0.00%
rguilayout	50654126	8.77%	91.25%	0.02%
rguistyler	63933807	8.62%	91.40%	0.02%
riconpacker	1078843	10.12%	90.10%	0.22%
rtexpacker	10	60.00%	100.00%	60.00%
rtexviewer	10	60.00%	100.00%	60.00%
sandspiel	300347531	18.65%	81.38%	0.03%
sqlgui	250114	39.27%	63.05%	2.32%
Geomean		**27.20%**	**38.17%**	**0.47%**

because multiplyDouble and multiplyInt do not perform any loads or calls during their execution, i.e., there is nothing for our techniques to optimize. Instead, all the events in the traces are `FuncEntry` events directly followed by `FuncExit` events. These functions use a loop internally within an exported function to repeat the multiplications. In contrast, the results for fib show that almost all trace events are filtered out, with only 24 out of 4.9 billion events remaining. This is because fib contains a recursive function that calls itself many times. Most of the events are `Call` events that get filtered out by the call stack optimization.

5.5 RQ4. Effectiveness of Replay Optimization

In this section, we evaluate the effectiveness of replay optimization techniques introduced in Section 3.4.3. We first conduct a simple comparison of the replay binary size of replay benchmarks 1) without any replay optimizations, 2) only with the memory write merge optimization, 3) only with the function split optimization, and 4) with both optimization techniques. We found that the function split optimization does not affect the binary size, while the memory write merge optimization reduces the replay binary size by 9.98%. Then, for the 27 replay benchmarks in Wasm-R3-Bench, we carried out an ablation study to evaluate the effectiveness of the two optimizations. We call this the *replay optimization experiment*. In the replay optimization experiment, we measure two kinds of times: *load and validation time* and *execution time*. The load and validation time is

Table 4. Replay optimization experiment results.

Name	**Load+Validation time (μs)**				**Execution time (μs)**			
	No	**Split**	**Merge**	**All**	**No**	**Split**	**Merge**	**All**
boa	242742.4	102.87%	108.98%	106.05%	47262.4	101.87%	104.92%	101.62%
bullet	27313.8	97.06%	100.47%	104.62%	358420.8	101.55%	113.76%	127.99%
commanderkeen	270972.2	116.73%	92.19%	111.92%	18778942.4	100.96%	97.95%	97.81%
factorial	2517.3	94.14%	96.90%	99.63%	64.7	97.84%	97.30%	99.85%
ffmpeg	264556.5	100.81%	113.45%	120.35%	22.8	101.32%	106.80%	107.68%
fib	5927.2	121.98%	118.52%	119.83%	58289332.4	99.99%	99.13%	98.15%
figma-startpage	10758.8	98.93%	101.38%	99.37%	63.8	99.22%	103.61%	98.75%
game-of-life	88.4	96.28%	94.02%	98.14%	37.6	98.88%	94.85%	100.52%
guiicons	17627.2	130.33%	113.69%	111.37%	522202.34	119.45%	120.20%	115.28%
hydro	31932.9	97.84%	99.85%	98.65%	84.6	89.24%	94.32%	90.54%
jqkungfu	29246.2	94.42%	90.05%	79.93%	23.6	98.09%	94.49%	88.77%
jsc	218154.5	102.24%	84.84%	87.97%	19046.9	97.36%	93.34%	96.51%
mandelbrot	60570.8	131.97%	7.26%	7.33%	44074249.6	99.55%	99.32%	98.91%
multiplyDouble	6533.8	101.58%	106.38%	107.45%	43857353.6	100.05%	99.26%	100.11%
multiplyInt	5861.4	120.96%	113.67%	120.45%	42531426.8	100.22%	99.16%	98.55%
pacalc	10414.5	90.26%	92.22%	103.58%	3453.8	92.95%	93.85%	102.23%
parquet	78150.8	90.79%	90.32%	90.08%	188.5	96.45%	96.07%	96.29%
pathfinding	21315.8	129.43%	123.20%	166.98%	3368370.8	101.60%	96.35%	85.21%
rfxgen	27742.7	98.81%	96.84%	94.44%	1146599.5	99.65%	98.32%	92.56%
rguilayout	26222.4	100.21%	98.27%	97.52%	440909.2	96.04%	98.48%	95.24%
rguistyler	21885.1	110.35%	114.29%	120.18%	469477.9	107.37%	120.39%	119.56%
riconpacker	25251.2	95.74%	84.12%	87.78%	16982.4	95.36%	87.32%	88.26%
rtexpacker	25133.0	87.92%	79.43%	78.43%	15.2	86.18%	86.18%	83.22%
rtexviewer	16457.3	97.52%	96.57%	88.02%	12.4	97.58%	94.76%	96.77%
sandspiel	22352.0	152.33%	141.32%	155.21%	3135717.0	105.89%	107.45%	92.61%
sqlgui	41837.3	103.06%	98.04%	98.30%	6790.0	100.99%	97.27%	97.28%
Geomean		**105.30%**	**91.35%**	**94.05%**		**99.28%**	**99.48%**	**98.40%**

the time spent loading, parsing, and validating the replay Wasm benchmark, while execution time represents time to execute the main (i.e. top-level replay) function. We measure these with the `--metrics` option of the Wizard engine, which reports load, validation, compilation, and execution time, and average over 10 runs.

Table 4 reports the results of the replay optimization experiment. On average, applying both replay optimizations reduces the time spent in load and validation by about 6%. In detail, applying the function split optimization increases the load and validation time by about 5%. Considering that some replay benchmarks require the function split optimization to run, we think this is an acceptable increase. Applying only the memory write merge optimization decreases the load time about 9%. Individually, mandelbrot seems to enjoy the greatest benefit, with the load and validation time reduced to just 7.33% of the original time. rtexpacker, jqkungfu, and riconpacker are other beneficiaries, with the load and validation time reduced to 78.43%, 79.93%, and 87.78% of the original load and validation time, respectively. On average, applying both replay optimizations decreases the execution time to 98.40%. We believe that although the replay optimizations do not significantly affect the execution time, it gives a slight performance improvement. We are currently investigating the reasons behind.

Note that funky-kart, despite successfully completing the experiment, is excluded from the table. This is because its replay benchmarks without replay optimizations exceeded the maximum size limit for function bodies imposed by the production Wasm engines, which prevented them from running on any engines. Thus, we discuss the performance characteristics of FUNKY-KART here. While the function split optimization makes the replay benchmark executable, it does not affect its binary size. Applying the memory write merge optimization on top of the function split optimization reduces the load and validate time to 7.80% and the execution time to 86.31% compared to the replay benchmark with only the function split optimization applied. Thus, alongside MANDELBROT, FUNKY-KART is another example where the memory write merge optimization significantly reduces the load time and validation time of the replay benchmark, with an added benefit of reducing the execution time.

In summary, we show that the replay optimization techniques play a crucial role in reducing the load time and validation time of performance outliers in Wasm-R3-Bench, while having a modest effect on the execution time.

6 Limitations

Currently, our Wasabi-based instrumentation supports Wasm version 2.0, with the exception of the SIMD proposal. This limitation can be easily overcome by using a more up-to-date instrumentation library. Among the proposals in phase 4, which are planned to be standardized soon, the multi-memory proposal is already supported by Wasm-R3. We also believe our approach can support Wasm GC with some help from the host environment. Our technique relies on making shadow copies of all mutable state and detecting modifications by comparing the shared state with the shadow state which necessitates host support, since Wasm `funcref` and `externref` do not have native Wasm comparison operators. The most challenging proposal to support would be threads, which remains an open research question, as deterministic replay of racy programs lacks satisfactory and robust solutions.

7 Related work

Record and replay is a mature area of research that has been explored in various contexts, including architectural support for record and replay [56], operating system-level record and replay [18, 24], and language runtime-level record and replay [49]. Typically, such systems require intrusive modification to the respective CPU design, kernel, libraries, or language runtime to record events, but can also be done by bytecode rewriting [29]. In any case, potentially non-deterministic operations and side-effects must be identified and recorded as events. Enumerating these operations can represent significant manual work. Our approach has a different requirement in that we aim to run Wasm-R3 on any architecture, operating system, or language runtime without modification. This requirement led us to using bytecode-level instrumentation.

Among previous instrumentation of this kind in the literature, Wasm-R3 is most similar to JSBench [45], a record and replay technique for the automated construction of JavaScript benchmarks. The major difference between JSBench and Wasm-R3 stems from the design differences in the languages they target: JavaScript and Wasm, respectively. In Wasm, instances and their host environment are cleanly separated by the import/export boundary. Thus, efficiently tracking non-determinism caused by the host environment boils down to making shadow copies of the shared mutable states and comparing them. In contrast, nearly any JavaScript operator could have unbounded side-effects. For instance, JSBench notes that a JavaScript `for...in` loop can be a potential source of non-determinism. This leads the authors of JSBench to describe their catalogue of non-determinism in JavaScript applications as "necessarily incomplete." We believe that the simplicity of Wasm-R3's record phase, in comparison to JSBench, is not a drawback but an advantage, demonstrating Wasm's suitability as a target for automatic benchmark generation for the web.

Jalangi [50] is another record and replay framework for JavaScript, designed for heavy-weight dynamic analysis. While Jalangi facilitates record and replay across different environments (e.g., recording on a mobile environment and replaying on a desktop environment), it operates within the confines of JavaScript engines, which accept the same inputs and produce identical outputs. This means that, despite the convenience offered by Jalangi's record and replay functionality, it does not enable something fundamentally impossible; it is possible to have the same execution in different environments by simply running the same JavaScript code. In contrast, Wasm-R3 enables the capability to replay a Wasm instance interacting with JavaScript code in non-web Wasm engines, which is impossible without Wasm-R3. Moreover, the basic technique of Wasm-R3 could allow recording in non-web Wasm engines and replaying in web Wasm engines. Another important difference is that, unlike Wasm-R3, Jalangi applies code instrumentation in both the record and replay phases to implement the replay of the recorded trace. Although this eliminates the need to precisely determine at which point in the host code an interaction occurred, which Wasm-R3 meets, it has the disadvantage of mixing up the original code and replay code. We instead aim to preserve the original binary and its exact instruction-by-instruction behavior and only replace calls to the host with replays. Lastly, Jalangi explicitly states that they did not make any effort to optimize their implementation. In contrast, we applied numerous optimizations to the trace to enable Wasm-R3 to scale to real-world applications, and to the replay IR, to ensure the resulting replay is representative of the recorded execution.

Other record and replay frameworks for JavaScript include Mugshot [39] and WebRR [36]. Mugshot [39] records browser events and shares our goal of recording on unmodified browsers. Unlike Wasm-R3, Mugshot is tightly coupled with the web browser implementations and adopts different strategies depending on the browser. In contrast, Wasm-R3 exploits the host-agnostic nature of Wasm to record the execution of Wasm applications across all browsers. WebRR [36] proposes a record and replay technique that enhances the robustness of fragile end-to-end tests. Their primary focus is on avoiding test failures that occur without a bug or misbehavior in the application under test. In contrast, benchmarks generated from Wasm-R3 do not suffer from any kind of fragility that plagues end-to-end tests. We refer to a comprehensive survey for a more detailed discussion of dynamic analysis for JavaScript [9].

Beyond record and replay, a lot of research has gone into Wasm. Hilbig et al. [28] proposed benchmarks of real-world binaries. Unlike the benchmarks created with Wasm-R3, their benchmarks are not executable, making them unusable for performance benchmarking. Several general-purpose techniques to dynamically analyze Wasm have been proposed, e.g., based on source-to-source instrumentation [32] or via dynamic instrumentation inside a Wasm engine [54]. Other work supports reverse engineering by inferring types [33] and the purpose of functions [48] in Wasm binaries, studies security issues in Wasm [31], and shows how to use Wasm for obfuscation [47]. Our work contributes to the field by providing the first record and replay technique for Wasm and a benchmark suite of executable Wasm binaries.

8 Conclusion

We present Wasm-R3, the first record and replay framework for Wasm. The approach works with no modifications to the source compiler, virtual machine, host environment, operating system, or hardware. During arbitrary execution of a Wasm module within a host environment, Wasm-R3 records all interactions with the host environment, detecting updates to shared mutable memory via a shadow memory. The resulting trace is optimized to produce a precise replay binary that reproduces the original program's behavior by replaying all interactions with the host. Wasm-R3 is broadly applicable to numerous real-world Wasm applications, and the replay files it generates can be run across both web and non-web Wasm engines. Several optimizations applied at both recording and replay make infeasibly long traces tractable and reduce the overhead of trace replay for more faithful execution characteristics. We have made Wasm-R3 available as open source and hope it will be beneficial to Wasm application developers to create repeatable replays of specific executions for benchmarking and other purposes. In particular, we hope the technique can unlock a new era of Wasm benchmarking that better represents real-world use cases by routinely generating replays from real applications in their respective host environments.

Data-Availability Statement

The artifact is available on Zenodo at [11]. It includes Wasm-R3 and the Wasm-R3-Bench benchmark suite.

Acknowledgments

This work was supported by the European Research Council (ERC, grant agreement 851895), and by the German Research Foundation within the ConcSys, DeMoCo, and QPTest projects. Also, this work was partly supported by the National Research Foundation of Korea (NRF) (2022R1A2C2003660 and 2021R1A5A1021944) and Institute for Information & Communication Technology Planning & Evaluation (IITP) funded by the Korean government MSIT (No. RS-2024-00337703). Partial support also provided by the National Science Foundation under award #2148301.

References

[1] 2018. TurboFan: V8's Optimizing Compiler. https://v8.dev/docs/turbofan. https://v8.dev/docs/turbofan (Accessed 2021-07-29).

[2] 2021. JavaScriptCore, the built-in JavaScript engine for WebKit. https://trac.webkit.org/wiki/JavaScriptCore. https://trac.webkit.org/wiki/JavaScriptCore (Accessed 2021-07-29).

[3] 2021. SpiderMonkey: Mozilla's JavaScript and WebAssembly engine. https://spidermonkey.dev. https://spidermonkey.dev (Accessed 2021-07-29).

[4] 2021. V8 Development Site. https://v8.dev. https://v8.dev (Accessed 2021-07-29).

[5] 2021. Wasmer: A Fast and Secure WebAssembly Runtime. https://github.com/wasmerio/wasmer. https://github.com/wasmerio/wasmer (Accessed 2021-07-06).

[6] 2021. Wasmtime: a standalone runtime for WebAssembly. https://github.com/bytecodealliance/wasmtime. https://github.com/bytecodealliance/wasmtime (Accessed 2021-08-11).

[7] 2024. *Announcing Speedometer 3.0: A Shared Browser Benchmark for Web Application Responsiveness.* https://browserbench.org/announcements/speedometer3/

[8] 2024. binjgb. https://binji.github.io/binjgb/. Retrieved April 5th, 2024.

[9] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. 2017. A Survey of Dynamic Analysis and Test Generation for JavaScript. *ACM Comput. Surv.* 50, 5 (2017), 66:1–66:36. https://doi.org/10.1145/3106739

[10] Clemens Backes. 2018. Liftoff: a new baseline compiler for WebAssembly in V8. https://v8.dev/blog/liftoff Accessed: 2024-03-08.

[11] Doehyun Baek, Jakob Getz, and Yusung Sim. 2024. *Wasm-R3: Record-Reduce-Replay for Realistic and Standalone WebAssembly Benchmarks (Artifact).* https://doi.org/10.5281/zenodo.13382344

[12] S. M. Blackburn et al. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*. 169–190.

[13] Derek L. Bruening. 2004. *Efficient, transparent, and comprehensive runtime code manipulation.* Ph. D. Dissertation. USA. AAI0807735.

[14] Bytecode Alliance. 2023. WASI: The WebAssembly System Interface. wasi.dev. https://wasi.dev/ (Accessed 2024-3-26).

[15] Gabriel Cuvillier. 2024. Arxwasm. https://wasm.continuation-labs.com/arxdemo/. Retrieved April 5th, 2024.

[16] Gabriel Cuvillier. 2024. D3wasm. https://wasm.continuation-labs.com/d3demo/. Retrieved April 5th, 2024.

[17] Frank Denis. 2021. LibSodium WebAssembly Benchmarks. https://github.com/jedisct1/webassembly-benchmarks. https://github.com/jedisct1/webassembly-benchmarks (Accessed 2023-5-7).

[18] George W Dunlap, Samuel T King, Sukru Cinar, Murtaza A Basrai, and Peter M Chen. 2002. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review* 36, SI (2002), 211–224.

[19] Aaron Turner et al. 2024. Wasmboy. https://wasmboy.app/. Retrieved April 5th, 2024.

[20] Matteo Basso et al. 2024. Awesome-Wasm. https://github.com/mbasso/awesome-wasm. Retrieved April 5th, 2024.

[21] Zakai et al. 2024. Binaryen. https://github.com/WebAssembly/binaryen. Retrieved April 3, 2024.

[22] Phani Kishore Gadepalli, Sean P. McBride, Gregor Peach, L. Cherkasova, and Gabriel Parmer. 2020. Sledge: a Serverless-first, Light-weight Wasm Runtime for the Edge. *Proceedings of the 21st International Middleware Conference* (2020). https://api.semanticscholar.org/CorpusID:228085728

[23] Google Chrome Developers. 2024. Chrome DevTools Protocol. https://chromedevtools.github.io/devtools-protocol/.

[24] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. 2008. R2: an application-level kernel for record and replay. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 193–208.

[25] Giuseppe Gurgone and Philipp Spiess. 2018. *A Real-World WebAssembly Benchmark*. https://pspdfkit.com/blog/2018/a-real-world-webassembly-benchmark/

[26] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*. 3–14. https://doi.org/10.1109/WWC.2001.990739

[27] 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. *SIGPLAN Not.* 52, 6 (jun 2017), 185–200. https://doi.org/10.1145/3140587.3062363

[28] Aaron Hilbig, Daniel Lehmann, and Michael Pradel. 2021. An Empirical Study of Real-World WebAssembly Binaries: Security, Languages, Use Cases. In *Proceedings of the Web Conference 2021* (Ljubljana, Slovenia) *(WWW '21)*. Association for Computing Machinery, New York, NY, USA, 2696–2708. https://doi.org/10.1145/3442381.3450138

[29] Yongjian Hu, Tanzirul Azim, and Iulian Neamtiu. 2015. Versatile yet lightweight record-and-replay for Android. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 349–366. https://doi.org/10.1145/2814270.2814320

[30] 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 19)*. 107–120.

[31] Daniel Lehmann, Johannes Kinder, and Michael Pradel. 2020. Everything Old is New Again: Binary Security of WebAssembly. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 217–234. https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann

[32] Daniel Lehmann and Michael Pradel. 2019. Wasabi: A Framework for Dynamically Analyzing WebAssembly. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 1045–1058. https://doi.org/10.1145/3297858.3304068

[33] Daniel Lehmann and Michael Pradel. 2022. Finding the dwarf: recovering precise types from WebAssembly binaries. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 410–425. https://doi.org/10.1145/3519939.3523449

[34] Friedrich Leisch and Evgenia Dimitriadou. 2010. *mlbench: Machine Learning Benchmark Problems*. R package version 2.1-1.

[35] Borui Li, Hongchang Fan, Yi Gao, and Wei Dong. 2022. Bringing webassembly to resource-constrained iot devices for seamless device-cloud integration. *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services* (2022). https://api.semanticscholar.org/CorpusID:249705610

[36] Zhenyue Long, Guoquan Wu, Xiaojiang Chen, Wei Chen, and Jun Wei. 2020. WebRR: self-replay enhanced robust record/replay for web application testing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1498–1508. https://doi.org/10.1145/3368089.3417069

[37] William Mbotta. 2024. Tic tac toe Wasm. https://sepiropht.github.io/tic-tac-toe-wasm/. Retrieved April 5th, 2024.

[38] Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. 2022. WebAssembly as a Common Layer for the Cloud-edge Continuum. *Proceedings of the 2nd Workshop on Flexible Resource and Application Management on the Edge* (2022). https://api.semanticscholar.org/CorpusID:249960276

[39] James Mickens, Jeremy Elson, and Jon Howell. 2010. Mugshot: deterministic capture and replay for Javascript applications. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation* (San Jose, California) *(NSDI'10)*. USENIX Association, USA, 11.

[40] Microsoft. 2024. Playwright. https://playwright.dev/. Retrieved April 5th, 2024.

[41] Matthias Müller, Brian Whitney, Robert Henschel, and Kalyan Kumaran. 2011. *SPEC Benchmarks*. Springer US, Boston, MA, 1886–1893. https://doi.org/10.1007/978-0-387-09766-4_370

[42] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Not.* 42, 6 (jun 2007), 89–100. https://doi.org/10.1145/1273442.1250746

[43] Harish Patil, Cristiano L. Pereira, Mack Stallcup, Gregory Lueck, and James H. Cownie. 2010. PinPlay: a framework for deterministic replay and reproducible analysis of parallel programs. In *IEEE/ACM International Symposium on Code Generation and Optimization*. https://api.semanticscholar.org/CorpusID:17445756

[44] Matthias J. Reisinger. 2016. PolyBenchC. https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1. https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1 (Accessed 2023-5-7).

[45] Gregor Richards, Andreas Gal, Brendan Eich, and Jan Vitek. 2011. Automated Construction of JavaScript Benchmarks. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 677–694. https://doi.org/10.1145/2048066.2048119

[46] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. 2010. An analysis of the dynamic behavior of JavaScript programs. In *ACM-SIGPLAN Symposium on Programming Language Design and Implementation*. https://api.semanticscholar.org/CorpusID:2334122

[47] Alan Romano, Daniel Lehmann, Michael Pradel, and Weihang Wang. 2022. Wobfuscator: Obfuscating JavaScript Malware via Opportunistic Translation to WebAssembly. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 1574–1589. https://doi.org/10.1109/SP46214.2022.9833626

[48] Alan Romano and Weihang Wang. 2023. Automated WebAssembly Function Purpose Identification With Semantics-Aware Analysis. In *Proceedings of the ACM Web Conference 2023, WWW 2023, Austin, TX, USA, 30 April 2023 - 4 May 2023*, Ying Ding, Jie Tang, Juan F. Sequeda, Lora Aroyo, Carlos Castillo, and Geert-Jan Houben (Eds.). ACM, 2885–2894. https://doi.org/10.1145/3543507.3583235

[49] Onur Sahin, Assel Aliyeva, Hariharan Mathavan, Ayse Coskun, and Manuel Egele. 2019. RANDR: Record and Replay for Android Applications via Targeted Runtime Instrumentation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 128–138. https://doi.org/10.1109/ASE.2019.00022

[50] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 488–498. https://doi.org/10.1145/2491411.2491447

[51] Ben L. Titzer. 2021. Wizard, An advanced WebAssembly Engine for Research. https://github.com/titzer/wizard-engine. https://github.com/titzer/wizard-engine Retrieved Februar 23, 2024.

[52] Ben L. Titzer. 2022. A fast in-place interpreter for WebAssembly. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 148 (oct 2022), 27 pages. https://doi.org/10.1145/3563311

[53] Ben L. Titzer. 2024. Whose Baseline Compiler Is It Anyway?. In *International Symposium on Code Generation and Optimization (CGO)*.

[54] Ben L Titzer, Elizabeth Gilbert, Bradley Wei Jie Teo, Yash Anand, Kazuyuki Takayama, and Heather Miller. 2024. Flexible Non-intrusive Dynamic Instrumentation for WebAssembly. *arXiv preprint arXiv:2403.07973* (2024).

[55] Aaron Turner, James Milner, and Jonathan Beri. 2024. Made with WebAssembly. https://madewithwebassembly.com/. Retrieved April 5th, 2024.

[56] Min Xu, Rastislav Bodik, and Mark D. Hill. 2003. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. *SIGARCH Comput. Archit. News* 31, 2 (may 2003), 122–135. https://doi.org/10.1145/871656.859633

Received 2024-04-06; accepted 2024-08-18