Software Lab
Institute of Software Engineering
University of Stuttgart
Universitätsstraße 38, 70569 Stuttgart

Master Thesis

Wasm-R3: Creating Executable Benchmarks of WebAssembly Binaries via Record-Reduce-Replay

Jakob Getz

Course of study:	Computer Science
Examiner:	Prof. Dr. Michael Pradel
Supervisor:	Sukyong Ryu, Ben L. Titzer, Daniel Lehmann, Doehyun Baek, Yuesung Sim
Started:	September 1, 2023
Completed:	March 1, 2024

University of Stuttgart
Germany

SOLA
SoftwareLab

Abstract

WebAssembly is the newest language to arrive on the web and has now been implemented in all major browsers for several years. It features a compact binary format, making it fast to be loaded, decoded and run. To evaluate and improve the performance of WebAssembly engines, relevant executable benchmarks are required. Existing benchmarks such as PolyBenchC and Spec CPU have shortcomings in their relevancy, since they do not necessarily represent real-world WebAssembly applications well. To make the creation of such benchmarks faster and simpler, we develop Wasm-R3 an approach that has the capability of recording existing web applications and generate an executable benchmark from it. Wasm-R3's workflow can be described in three phases: record, reduce and replay. In the record phase the instrumenter instruments the website's WebAssembly code, a user then interacts with the website, which causes traces of the execution to be recorded. Since these traces are typically large, unnecessary information gets filtered out in the reduce phase. In the replay phase a replay generator takes these traces along with the original web application's WebAssembly binary and generates a standalone executable benchmark from it. We evaluate Wasm-R3 by implementing it in Typescript and Rust to show that the generated benchmarks correctly mimic the behavior of the recorded application. We further demonstrate that replays can be generated in reasonable time by measuring a mean wall time of 8.651 seconds and that our benchmarks are portable across a variety of different WebAssembly engines.

Zusammenfassung

WebAssembly ist die neueste Webtechnologie und ist seit mehreren Jahren in allen großen Browsern integriert. WebAssembly ist ein kompaktes Bytecode Format, welches schnell geladen, decodiert und ausgeführt werden kann. Um die Performance von WebAssembly Engines zu evaluieren und zu verbessern, sind relevante und ausführbare Benchmarks notwendig. Existierende Benchmarks wie PolyBenchC und Spec CPU haben Mängel in ihrer Relevanz, da sie nicht zwangsläufigerweise Anwendungen der echten Welt repräsentieren. Um das Erstellen solcher Benchmarks einfacher und schneller zu gestalten, entwickeln wir Wasm-R3, ein Werkzeug welches die Fähigkeit besitzt, existierende Webanwendungen aufzunehmen und durch die Aufnahme eine ausführbare Benchmark zu generieren. Wasm-R3s Arbeitsablauf kann in drei Phasen beschrieben werden: Record, Reduce und Replay. In der Record Phase ist ein Instrumenter verantwortlich den WebAssembly Code der Webseite zu instrumentieren. Ein Benutzer interagiert anschließend mit dieser Webseite was dazu führt, dass Traces von dieser Ausführung aufgenommen werden. Da diese Traces typischerweise sehr groß sind, werden unnötige Informationen in der Reduce Phase heraus gefiltert. In der Replay Phase nimmt ein Replay Generator diese Traces zusammen mit der originalen WebAssembly Binary als Eingabe und generiert daraus alleinstehende Benchmarks. Wir evaluieren unseren Ansatz indem wir ihn in Typescript und Rust implementieren um zu zeigen, dass die generierten Benchmarks das Verhalten der originalen Anwendung korrekt wiedergeben. Des Weiteren zeigen wir, dass das ausführen von Wasm-R3 in einem vernünftigen Zeitrahmen von durchschnittlich 8,651 Sekunden stattfindet und dass unsere Benchmarks von einer großen Vielfalt an verschiedenen WebAssembly Engines ausgeführt werden können.

Contents

1 Introduction **1**

2 Record and Replay **5**

2.1 Classification of Record and Replay Systems 6

2.2 A Basic Framework 6

2.2.1 Environment and Application 6

2.2.2 Structure of the Environment 7

2.2.3 Store 7

2.3 Collecting Runtime Information 7

2.4 Generating Replays 8

3 WebAssembly **9**

3.1 Structure 9

3.2 Computational Model 10

3.3 Execution Semantics and Nondeterminism 10

4 Approach **13**

4.1 Architecture 13

4.2 Record 15

4.2.1 Required Data to Record 15

4.2.2 Naive Approach 16

4.2.3 Decoupled Recording 16

4.2.4 Trace Structure 17

4.2.5 Trace Implementation 19

4.2.6 Instrumentation Implementation 19

4.2.7 Trace Transfer Implementation 21

4.2.8 Proxy Implementation 22

4.3 Reduce 23

4.3.1 Shadow Optimization 23

4.3.2 Call Stack Optimization 25

4.3.3 Implementation 27

4.4 Replay 27

4.4.1 Replay Intermediate Representation 28

4.4.2 Non Monotone Replay Generation . . . 29
4.4.3 Optimizations . . . 30
4.4.4 Backends and Output Formats . . . 30

5 Evaluation **35**
5.1 Experimental Setup . . . 35
5.2 Effectiveness of Recording (RQ 1) . . . 37
5.3 Effectiveness of Reduction (RQ 2) . . . 37
5.4 Portability of Wasm-R3 (RQ 3) . . . 39
5.5 Correctness of Wasm-R3 (RQ 4) . . . 40
5.6 Performance of Wasm-R3 (RQ 4) . . . 40
5.6.1 Wall Time . . . 40
5.6.2 Recording Overhead . . . 41
5.6.3 Replay Generation . . . 42
5.6.4 Replay Execution Time . . . 43

6 Discussion **45**

7 Future Work **47**

8 Related Work **49**

9 Conclusion **51**

Bibliography **51**

1 Introduction

WebAssembly is a safe, portable, low-level code format designed for efficient execution and compact representation [2]. It is intended mainly to be used as a compilation target, for different source languages, including C, C++, Rust and Go, to make software written in those languages executable in the browser. Its low-level instructions map close to hardware instructions with the goal to achieve execution speeds close to native performance. Even though significantly faster then JavaScript and runnable through browsers engines such as Webkit or V8 it is not intended to replace JavaScript but instead augment it, which promises to speed up specific components of the broader application. Since its introductory paper by Haas et al. [18] WebAssembly has been gaining traction and got adopted by many major applications such as Figma[1] and CapCut[2].

Although WebAssembly is designed to be faster than JavaScript, real-world outcomes have varied [4, 7]. For instance, eBay developers experienced a significant performance boost by up to 50 times when they utilized WebAssembly to develop a barcode scanner [29], compared to their JavaScript implementation. Conversely, Samsung engineers found that on the Samsung Internet browser (version 7.2.10.12), WebAssembly was slower than JavaScript when conducting matrix multiplications of specific sizes [6]. To investigate similar ambiguous cases and support the development and improvement of virtual machines multiple studies with different conclusions have been conducted to analyze the performance of WebAssembly vs Javascript [37, 9, 10, 36], and native code [22, 23].

All of the above studies rely on benchmarks to collect performance data, which get typically run in an instrumented version of the virtual machine under investigation. In an experimental setup this, a careful selection of benchmarks is crucial to obtain representative and reliable data. Unfortunately, finding a sufficiently large set of representative WebAssembly benchmarks is challenging, since due to its young age only a small number of benchmarks exists. Lehmann et al. contribute a diverse suite of 8,461 real world wasm binaries, however these are not executable and thus not of use for performance measurement [24, 20]. This deficiency on WebAssembly benchmarks forces researchers to either write their own programs or use commonly used standard benchmark suites such as *PolyBenchC*, *CHStone* [19] or *Spec CPU* which are written in C and compile them to WebAssembly. While suitable for collecting minimal data, these benchmarks might not necessarily be representative of WebAssembly applications as they appear in the wild. More specifically the types of actions performed in industry standard benchmarks can vary greatly from the real world [30]. These biased benchmarks can lead language developers astray by promoting optimizations that are

[1]www.figma.com

[2]www.capcut.com

not actually significant in real-world usage and by overlooking areas where optimization could be beneficial. In the past, inadequate benchmarks have had a detrimental effect on the development of language implementations. For instance, the SPECjvm98 benchmark suite was widely utilized for assessing Java performance despite widespread acknowledgment within the community that it didn't accurately represent real world applications [11]. This dissatisfaction ultimately prompted the development of the *DaCapo* benchmark suite, which features more realistic programs [3].

Creating benchmarks that are representative and exemplary of real-world applications is a difficult problem. A good strategy on solving this, is to use actual real-world applications as a starting point for benchmarking. One such WebAssembly benchmark is the *pspdf kit* online benchmark, which measures the runtime of different actions of a PDF library [17]. Despite being well crafted, it takes a lot of engineering effort to create and maintain these benchmarks, which is partially the reason why there is just a limited number available for WebAssembly so far. In addition such benchmarks are not portable and run only on the platforms they have been developed for.

To overcome these problems we develop Wasm-R3 an approach to record and replay WebAssembly web applications to quickly and reliably create benchmarks that represent real-world scenarios. We took conceptual inspiration from Richards et al. and their tool JSBench which applies a similar idea to create JavaScript benchmarks [30]. Being similar in spirit, this work however has to cope with very different challenges and uses very distinct principles and techniques from JSBench. Using a record and replay approach has a variety of advantages. Replays may not only be used as benchmarks but also as a mean to debug faulty applications, investigate security issues, or to perform other kinds of dynamic analysis. A variety of previous work has used a record and replay approach to implement different debugging tools [32, 27, 34]. Other papers investigate its potential to improve the security of applications [14, 13].

This thesis is structured into 8 chapters. In Chapter 2 we will describe record and replay systems in general, by introducing a conceptual framework featuring the concepts of application and environment. Knowledge we gather in this part will help to later understand the the structure of Wasm-R3. Chapter 3 provides an background on WebAssembly and highlights its components that influence the design of a corresponding record and replay system. Chapter 4 pictures our approach by describing its three elemental components record, reduce, replay and providing details about our implementation. We evaluate our approach in Chapter 5 and argue for its applicability in real world scenarios. Later chapters will discuss our findings, present future work and conclude the topic.

In summary this thesis contributes the following:

- We introduce the first record and replay system for WebAssembly web applications for creating benchmarks of real world applications with low effort.
- We present techniques to address unique technical challenges not present in existent record and replay systems, including reducing the recorded trace through shadow optimization and call stack optimization, as well as non monotone replay generation.

- We show that Wasm-R3 is effective in-real world scenarios by applying it to 17 real world web applications, verifying its correctness, as well as measuring a variety of different metrics such as wall time, replay generation time, and record performance.
- We implement Wasm-R3 and make it available as open source.

2 Record and Replay

Software typically contains nonpure behavior. We call behavior nonpure when the execution and output of a particular program depends on state and is independent of the running process. As an example a program may call an API for retrieving current information about the weather. The behavior of the program will depend on the data it retrieves from that API. In addition, a program's behavior often depends on user provided inputs, that can interrupt the usual execution and trigger a different behavior. From an application perspective nonpurity and the ability for user input to alter the program's execution expresses itself as nondeterminism. A program is nondeterministic when we cannot predict its behavior by just obtaining static information about the program and its initial state. This property of most applications signifies that the execution of the same program might yield different results for separate runs.

There are several applications where this trait leads to shortcomings and restrictions. Consider for example a bug that only appears with a certain probability in a nondeterministic fashion. To eliminate this bug one must locate it first by narrowing down to the possible locations it may appear. This process is significantly complicated if a developer is not able to faithfully reproduce the bug and instead relies on chance to observe it. Such problems motivated a category of techniques which try to record and replay the execution of a program to execute it deterministically with arbitrary repetitions. Record and replay techniques have been extensively studied in the past, mostly for debugging purposes [28, 5] and security analysis [15]. However this is not the only scenario in which such approaches could be applied to resolve a problem. The work of Richards et al. [30] represents one of the rare exceptions in that record and replay techniques are used for a different objective, namely the creation of javascript benchmarks, derived from real world browser applications. These benchmarks are intended to be used by engine developers, to improve on the performance on their implementation and not be reliant on artificially created benchmarks, that in many cases do not accurately represent real world applications and thus may lead to the implementation of optimizations that are not relevant or even harmful for most real world scenarios. Replay based debuggers have been surveyed and categorized by Dionne et al. [12] and Cornelis et al [8]. In this thesis we use especially the taxonomy of the latter to describe the underlying concepts and constraints of replay systems.

This chapter introduces the basic concepts behind record and replay systems. In Section 2.1 we describe the approaches content-based and ordering-based. Section 2.2 introduces the concept of environment and application, a framework for describing record and replay systems. Section 2.3 explains which information needs to be traced to create replays. The generation of these replays is described in Section 2.4.

2.1 Classification of Record and Replay Systems

All record and replay approaches rely on a trace which collects input and output values during the initial runtime of a specific program and provides the necessary information during replay to repeat the original execution. On the lowest level programs can be viewed as streams of instructions each of which consumes input and produces output. To produce a faithful replay of such a program a naive approach would trace the input and output of each instruction and provide these values to the same instructions during replay. This content-based approach has the advantage that a replay may be started and stopped at any program point while always preserving the original semantics. Despite this feature this approach is not feasible in practice since tracing of all inputs and outputs of each instruction would result in enormous trace sizes. Another approach is based on the idea of capturing the initial state of the recording execution as complete as possible and rerun the program based on that starting point. As the program leaves its initial path during re-executing due to nondeterminism, the process will be nudged to the desired path through information that was additionally captured during the record phase. This approach is called ordering-based. While implementable in practice this technique has the significant disadvantage over content-based approaches in that the replay can only be started at the initial state which was captured during the first execution. Abstracting from the instruction level, record and replay can also be applied to more high level programs using more procedural languages instead of machine instructions.

2.2 A Basic Framework

The following section introduces a basic framework to describe record and replay systems and precisely define them.

2.2.1 Environment and Application

Nondeterminism gets introduced to a program by its surrounding. Interrupts such as commands by peripheral input devices or dedicated input instructions that demand information from the surrounding system may cause the program to diverge from its usual path. More formally we can describe the execution of an application through the interaction between the deterministic application A itself and the surrounding environment E. In A's perspective E appears as a black box that provides unpredictable values at unpredictable timings to which A reacts in a well defined manner. Specifically, there are two types of interactions possible. 1. A may call E to receive information that it can not compute by itself and 2. E might send signals or interrupts to A whose parameters can be arbitrary.

As example we can concretize this concept by choosing A to be a user space application and E an operating system. The application will request certain services from the operating system such as reading a file from the filesystem via its system call interface and retrieve certain information about the outcome of these actions to which it reacts accordingly. Similarly the operating system can send interrupts to the application when the user uses the keyboard or mouse to issue certain commands which also get handled by the application. To create a faithful replay for the execution of

A, one must record the full information exchange between A and E that triggers certain behaviors in A. This information can later be used to modify E in a way to behave deterministically at every re-execution of the program.

2.2.2 Structure of the Environment

The nature of E is such that it is divided into layers. Each layer communicates via interfaces with the layer directly above and below it. We call the highest level layer wich provides an interface to A *first order environment* E', the layer directly below it *second order environment* E'', and so on. All $E', E'', ..., E^{(n)}$ (where $E^{(n)}$ denotes E with n primes) make up the complete environment E. To make this more clear we again consider the above example. Previously we have been representing E through the operating system. In fact the operating system, which represents the first order environment runs itself on the hardware, the second order environment with which it communicates via the instruction set architecture. As another example we can consider a javascript application which runs inside a web browser. The first order environment here is the browser itself or the virtual machine that powers the execution of the application code which itself runs on the operating system, which again runs on bare hardware.

2.2.3 Store

A factor that introduces further complication is the concept of a *store*. A store S hosts information which A either reads or writes during runtime. In case certain parts of this store are shared between application A and the environment E one must check, weither S changed during the execution of E's functions. Since the shared part of S can be quite large this in itself is a nontrivial task and the question arises how to exactly trace S's current state. Section 4.2 will evaluate on this problem further and offer solutions for the tracing of S in WebAssembly applications.

2.3 Collecting Runtime Information

To create a faithful replay of a program execution one must record all information exchanged between the application and the environment. The information typically gets collected in a trace that contains an entry for each relevant event that happens during program execution and its associated values. The system can then use this data during replay to provide the same information to A as they where observed during record. Traces of this kind can grow to large sizes which motivates its optimization and compression to keep the memory usage on a reasonable level.

To collect traces one must measure certain attributes of the program during runtime. In other words we require a method that is able to observe and safe the for replay relevant events. This goal can be achieved in two ways. Medicine distinguishes between measuring instruments that look inside a patient vs those that do not. Similarly we can decide between instrumentation on the source code level vs instrumentation inside of the environment. Some articles call these different approaches *invasive* and *non-invasive* [21], while others call them *intrusive* and *non-intrusive* respectively [16]. In the following we will stick to the latter terminology. To instrument an application intrusively an instrumented adds additional instructions to its source or byte code which observes and transfers the desired information ideally without altering the programs semantics.

2.4 Generating Replays

A replay generator will take the during recording of application A collected runtime information as input. It may also need static information about A to connect the generated replay properly to A's interfaces. The replay will be generated code that mimics or modifies the first order environment E' of E in a way that all of its interfaces are preserved and the original application A still performs the same calls to E' without being aware of the fact that E' might have been changed. It is sufficient to just change the first order environment E' of E, however depending on the specific scenario it can make sense to additionally or exclusively change E at one or multiple different layers. A replay can mimic E' by either *modifying*, *poxing*, *replacing*, or *ignoring* it.

Modifying: The replay generator will add additional logic to the already existing logic of the environment to alter its behavior in a way that makes it deterministic and predictable and behave in the same way as during recording. This approach requires access to E's code.

Proxying: As an alternative the environment can be proxies to intercept any cross function calls and making sure that these calls deliver the same information as the calls during record. An advantage of this method is that one does not need to access and mutate the environment which grands broader flexibility.

Replacing: By replacing the entire environments through a mock api that performs the desired actions the replay is made portable across different lower order environments. Replays are no longer restricted to be executed on the same architecture they have been recorded on.

Ignoring: A more intrusive approach lets the replay generator add additional code to A's original source code. This added code comes in the form of function definitions that replace the functions provided by E. The calls to the environment will get redirected to call one of the internally generated functions which implement the recorded behavior. This strategy follows a similar underlying concept as replacing and also decouples the replay from the environment it was recorded on.

3 WebAssembly

WebAssembly was first introduced to the scientific community 2017 by the conference paper *Bringing the Web up to Speed with WebAssembly* [18]. The bytecode format was designed by collaborating engineers of all major browser vendors google, microsoft, mozilla and apple. To this date WebAssembly is stably implemented in their javascript engines and its specification is under continuous development [1]. WebAssembly is designed to be fast. Being a compact bytecode its size is relatively small, which makes it fast to transfer over the wire. Furthermore the linear structure makes validation in a single pass possible. Validation can be done while streaming the bytecode from the internet. The instructions encoded in the bytecode map close to hardware instructions and thus guarantee high execution speeds.

Engine developers rely on benchmarks to compare and classify execution speeds and improve their engines performance. This chapter will describe the important properties of the WebAssembly standard for Wasm-R3. Section 3.1 goes over particularities of WebAssembly's structure. Section 3.2 talks about its computational model and what happens when WebAssembly code gets executed. Section 3.3 builds on top and describes WebAssembly's execution semantics and how nondeterminism is introduced to the program.

3.1 Structure

A WebAssembly binary is structured as a module, encompassing various entities such as function definitions, tables, linear memories, and global variables that can be either mutable or immutable. These entities can also be imported from other sources, specifying a module and name pair along with an appropriate type. Furthermore, each entities can be designated for export under one or more names, allowing them to be accessed externally. Apart from these entities, modules can include initialization data for their memories or tables. This data is represented as segments that are copied to specified offsets within the memory or table. Additionally, modules can define a start function, which is automatically executed when the module is loaded. In the following we present two important entity types of a WebAssembly module. There are other types that need to be considered by Wasm-R3, however we will omit their introduction for space reasons.

Functions: Every function in WebAssembly accepts a sequence of values as parameters and produces a sequence of values as results. These functions have the capability to call one another, including the ability to call themselves recursively. This recursive calling creates an implicit call stack, which is not directly accessible to the programmer. Additionally, functions within WebAssembly can declare local variables. These variables behave like virtual registers within the function, allowing for temporary storage and manipulation of data during the function's execution. These local variables are mutable, meaning their values can be changed as needed within the function's scope. Like other entities, functions may be imported from the host environment in which case a call to the respective function is effectively a call to the host environment. It may also be exported, which enables it to be called by the host environment.

Memories: WebAssembly's linear memory represents a continuous, changeable array of raw bytes. This memory is initially created with a set size but can expand or grow dynamically as needed during program execution. Within the program, it's possible to load from and store values into this linear memory at any byte address, even if the address is not aligned. When loading or storing integers into the linear memory, the program can specify a storage size that is smaller than the actual size of the integer type. However, if an access attempts to go beyond the bounds of the current memory size, a trap occurs. This trap serves as an error mechanism, indicating that the program has attempted an invalid memory access operation. When we apply the in Section 2.2 introduced framework on WebAssembly, entities such as memories, tables and globals are part of the store S.

3.2 Computational Model

The WebAssembly computational model relies on a stack machine design. When code is executed, it follows a sequence of instructions. These instructions work with values on a hidden operand stack and are divided into two main types. The first type, simple instructions, carry out fundamental operations on data. They take arguments from the operand stack, perform operations, and then place the results back onto the stack. The second type, control instructions, manage the flow of execution. This flow is organized, meaning it is represented using neatly nested structures such as blocks, loops, and conditionals. Branches within the code can only target these structured constructs, ensuring a clear and controlled program flow.

In addition to the operand stack there is the global store which a WebAssembly instruction may modify. This store consists out of WebAssembly's entities such as memories, tables and globals. If any of these entities contained in the store are either imported or exported we refer to them as public entities.

3.3 Execution Semantics and Nondeterminism

WebAssembly runs inside of an environment. If WebAssembly is part of a web application the first order environment is the JavaScript context. WebAssembly can call into this environment by calling functions imported from JavaScript, or being called by this environment by exposing its own functions via exports. The official WebAssembly specification defines the Wasm execution semantics in terms of reduction rules between configurations. Simplified, configurations consist out of the operand stack and the current state of the store. These reduction rules are strictly deterministic with the exception of the invocation of a host function. Such calls may either terminate with a trap or return regularly. In the latter case, the instruction will consume a fixed amount of arguments from the operand stack and push a fixed amount of results back. It may also modify the store in a nondeterministic manner. However it cannot remove and can only modify public entities.

I record and replay system for WebAssembly applications thus has to cope with calls to imported functions.

4 Approach

The following chapter introduces Wasm-R3, a record and replay system for WebAssembly web applications. In summary we can describe the underlying abstract approach of Wasm-R3 in three phases: record, reduce and replay. Figure 4.1 shows these three phases and their interactions. As depicted we execute a given WebAssembly application and generate a raw trace. During reduce unnecessary information gets filtered out of the raw trace to create an optimized trace for replay generation. The final phase replay generates the replay code that can be run together with the original target binary as an executable benchmark.

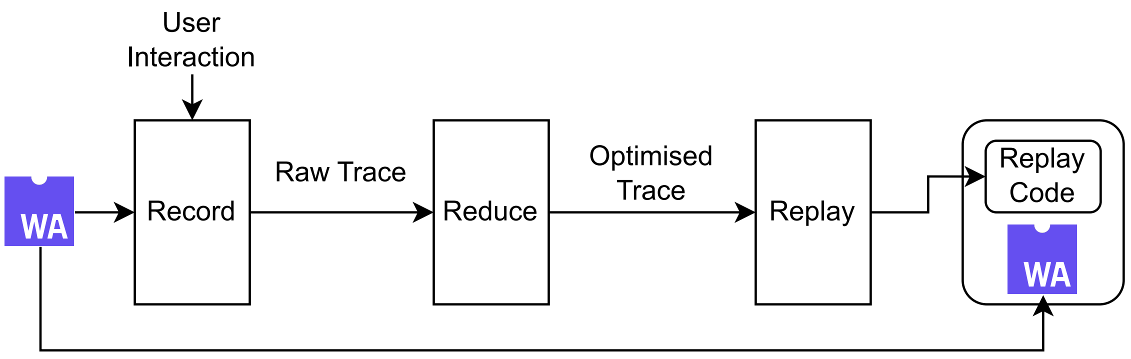


Figure 4.1: The three phases of Wasm-R3.

To introduce Wasm-R3 we will start in Section 4.1 by describing its abstract architecture. We continue by explaining the details behind its three phases. Record in Section 4.2, reduce in Section 4.3, and replay in Section 4.4. This chapter will focus on the underlying concepts and does not provide precise standards and definitions of the implemented datastructures, formats and algorithms. The precise and most up to date definitions will be available in the source code repository of this project.

4.1 Architecture

Instead of describing Wasm-R3 through its three phases, it can also be described through its components and their composition. This provides a more fine grained view on the system.

Given a URL, Wasm-R3 will automatically start the respective application in the browser. As a first step in the record phase, the **instrumenter** instruments all WebAssembly modules for the recording purpose. Then, when the application is loaded and user interaction begins, the **recorder** records the communication between WebAssembly and the host environment. The recorder produces traces, which hold the necessary information to precisely replay the interaction. In the replay phase, the **replay generator** reads the produced trace as well as the original WebAssembly binary and generates the replay IR. The replay IR is defined in terms of an intermediate representation and can be outputted through a **backend** in multiple concrete representations such as JavaScript or WebAssembly. We refer to these concrete representations as replay code. As a final step, the replay code is packaged with the web application's WebAssembly binary, being an executable and portable benchmark that represents the execution of the original web application. Since traces can get very large, they can be reduced in size through techniques which filter out a large chunk of potentially recorded events, which do not provide necessary information to create a deterministic and accurate replay of the original execution.

The concrete architecture for this design can be set up in a variety of different ways which will greatly affect performance and flexibility of the underlying tool. Common for all Wasm-R3 architectures is a proxy, which intercepts the web applications before being served to the browser and adds a recording runtime, which is responsible for (1) intercepting the calls to WebAssembly instantiation functions such as `WebAssembly.instantiate` to add intrusive instrumentation to the module's instructions and (2) provides a mean to represent the collected trace and the instantiated WebAssembly modules in the JavaScript context to send it at appropriate time to the server for further processing. The overview of this abstract architecture is depicted in Figure 4.2. The proxy and the recording runtime refer to the phase record of the abstract approach. The server then takes these traces as well as the original binaries to generate the final benchmarks. The work that is done by the server to generate the replays refers to the replay phase of the abstract approach. The phase reduce can happen at multiple different stages of the architecture. It can either happen synchronously during record or asynchronously on the server right before replay generation. Further sections will discuss different design decisions of this architecture and tradeoff between synchronous and asynchronous reduction in more detail.

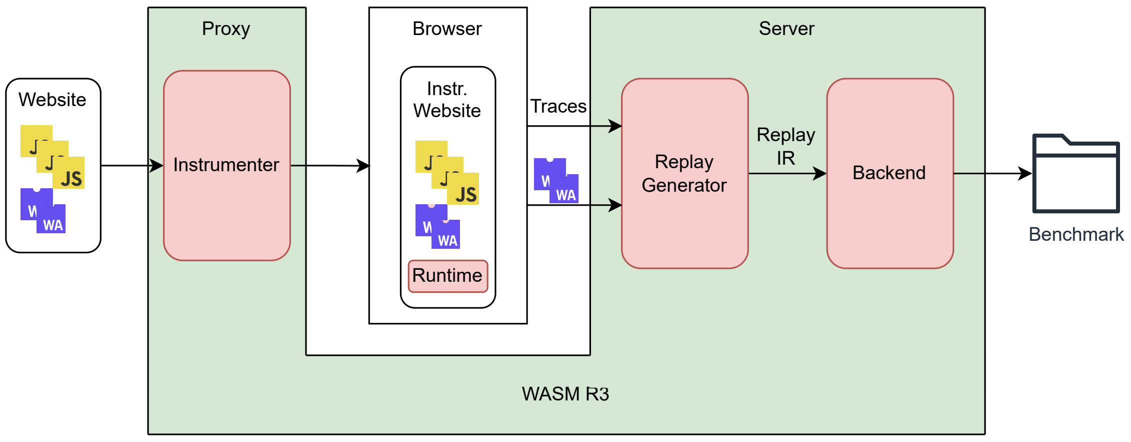


Figure 4.2: **Wasm-R3 System Architecture.** A proxy intercepts the website and adds the recording runtime to it. Traces and the original binaries get sent to the server which generates the final benchmarks.

During development we approached each of the three phases record, reduce, and replay in different ways. We implemented those different approaches in typescript and rust. Table 4.1 provides an overview on these implementations. The following sections will discuss their advantages and disadvantages in more detail.

Phase	**Option 1**	**Option 2**
Record	Wasabi based instrumentation	Custom instrumentation
Record	Bulk trace transfer	Streaming
Reduce	On the fly optimization	Asynchronous optimization
Replay	JavaScript based	Rust based

Table 4.1: An overview on the different implementation approaches to Wasm-R3's three phases

4.2 Record

This Section will describe our approach to obtain all the necessary runtime information from a WebAssembly program execution to create deterministic and accurate replays. Recording execution traces imposes challenges on performance and memory consumption of the record process and if not done thoughtful can lead to performance properties that disallow the usage of a specific recorder for specific applications. It is thus crucial to minimize the instrumentation overhead as well as the amount of data that is collected in a trace, since a rapidly growing trace will consume more memory space as there might be available. In Section 4.2.1 we describe the information that is necessary to collect in order to create a replay. Section 4.2.2 presents a naive approach that is able to collect this data, but lacks some important optimization. Section 4.2.3 finally explains the technique we use to generate the traces. Section 4.2.4 describes the structure of these traces. Its implementations are described in Section 4.2.5. Moving on, Section 4.2.6 describes our concrete approaches we took for implementing the instrumentation. Section 4.2.7 talks about the trace that lives inside of the web browser and how it is transferred to the server. We also elaborate on memory problems we face and how we solve them by using streaming. Finally, Section 4.2.8 discusses a critical part of Wasm-R3s's architecture that enables the recording of arbitrary applications in the web browser.

4.2.1 Required Data to Record

As described earlier in Section 3.3, nondeterminism gets introduced to WebAssembly applications by its functions imported from the host environment, which in the case of web applications are native JavaScript function calls. A recorder thus needs to capture information about these cross function calls. This includes any values returned by the respective functions and the functions effect on the store S. Entities of this store that can be affected by the host environment are globals, tables or memories that are either exported or imported by the WebAssembly module. We refer to these entities as public entities, as introduced in section 3.2. The recorder also needs to capture information about exported functions being called by the environment. This information contains provided parameters and the store configuration.

Listing 4.1: Example WebAssembly Module

```
1    (module
2        (import "env" "foo" (func $foo (param i32) (result i32)))
3        (func (export "entry") (param i32)
4            ;; instrumentation of function entry
5            ;; capturing of params and memory state
6            local.get 0
7            i32.load
8            call $foo
9            ;; instrumentation of call return
10           ;; capturing of results and memory state
11           drop
12       )
13       (memory (export "mem") 1)
14   )
```

Listing 4.1 instrumentation shows an example WebAssembly module. Line 3 shows the definition of an exported function. Wasm-R3 needs to capture the input parameters and the store configuration when this function is called. Line 8 shows the call to an exported function. Here the store changes as well as the return values need to be captured. The store in this example consists out of a memory, defined in Line 13. This memory is exported which makes it a public entity.

4.2.2 Naive Approach

Following a naive approach all information necessary can be collected by capturing the program state at every JavaScript to WebAssembly call and at every JavaScript to WebAssembly return. We refer to these locations in the code as function entry and call return. At these program points all parameters and return values have to be traced. In addition this approach would capture the complete state of all public entities, the store S. Listing 4.1 depicts this strategy. The instrumentation instructions for the function entry need to be inserted at line 4 while the instructions for the call return need to be inserted at line 9. While being simple, following this strategy comes with inefficiencies. Specifically, WebAssembly's memory can grow up to multiple gigabytes depending on the application, which recording of its full contents on each function entry and call return is impractical. It is hence required to use a optimized strategy to obtain a trace.

4.2.3 Decoupled Recording

We can observe that even though the full store of the WebAssembly program is available at all times, at most one unit of information inside of this store will be used per instruction. This means that it is possible that only a chunk of the full store influences the runtime of the program during its execution. As a consequence it is not mandatory to record the full store for generating a replay, it is instead more efficient to record only the parts of the store that are indeed read. In case of the example depicted in Listing 4.1 there are no instructions related to the defined memory which makes a tracing of the memory state redundant. This optimization can be implemented by

Listing 4.2: Type definition for the trace structure

```
Trace      = Vec<Event>
Event      = FuncEntry | FuncReturn | Call | CallReturn | Load
FuncEntry  = { funcidx: I32, params: Vec<ValType> }
Call       = { funcidx: I32 }
CallReturn = { funcidx: I32, results: Vec<ValType> }
Load       = { memidx: I32,
                 address: I32,
                 value: ValType | I8 | i16 }
ValType    = I32 | I64 | f32 | f64
```

recording only the values of the store when they are actually used by an instruction. We hence instrument not the function entries and call returns for observing values of the store but instead the respective instructions. If we want to, for example, record information about the state of the memory we can instrument all load instructions. Doing so will give us information on which value got loaded from which address. We call this technique decoupled recording.

This approach however has its challenge. So far we where recording all store information at immediately after every potential change. This means that the replay generator could read the trace linearly and map the trace entries succeedingly to the replay actions. No we record information about the store only at places when this information gets read by the WebAssembly program. By doing this we do not know at what specific point in time this piece of information as been modified by the environment. This makes replay generation more complicated. Section 4.4.2 will further discuss this newly introduced challenge, which we refer to as non monotone replay generation, and its algorithmic solution. It will further explain the underlying principles in more detail with an example.

4.2.4 Trace Structure

We define the trace structure, captured through decoupled recording, through a type definition. This type can have multiple concrete representations which may be used for in-memory representation, file storage, and debugging purposes. Listing 4.2 provides a definition of the trace structure type in pseudocode. For simplicity, we omit the definition of table get and global get events.

A trace is a sequence of events. The sequence might be empty in which case no execution of the underlying WebAssembly program has been taking place. Next to the above introduced events for function entry, call return and load we additionally add the events for function return which occurs at the end of a local WebAssembly function and call which occurs before a call instruction. These auxiliary events are used during replay generation to reorder the store related events such as load to guarantee the respective replay actions happen at the initially observed time in the host code. We describe its usages in Section 4.4.2. The function entry event holds next to the list of parameters the function index of the function that has been entered during recording of this event. A function index needs to be preserved also in the call and call return events to indicate which function actually got called by that instruction. Load events need to encode the loaded value in respect to the specific load instruction that WebAssembly provides. In WebAssembly, load instructions come in different flavors and can load a single byte or up to 64 bits at once. The value needs to be encoded depending on this instruction type. For example the event corresponding to `i32.load8_u` only needs to save one byte while `i64.load` saves eight. The address needs to hold the input address of the instruction summed up with the static offset that can be provided as an immediate value.

To record all possible nondeterministic behavior of a module in the real world more instructions than the ones listed in 4.2 are needed. Similar to recording loads, instructions related to public tables and globals need to be instrumented. These include but are not limited to `call_indirect` and `global.get`, which will record table get and global get events. Wasm-R3's implementation also handles these events.

The abstract syntax of the trace can have multiple concrete representations. For the purpose of this report we will use a JSON like textual representation of the trace, which will be reused throughout different figures. This representation will be only used to illustrate the discussed concepts and thus will not be formally defined. To depict a event we will start with the entry type followed by the related information as key-value pairs enclosed by curly braces. Vectors will be enclosed by square brackets. Number types such as `I32` will be written like this: `I32(0)`, however these will be optional and can sometimes be inferred implicitly. Sometimes fields that are not relevant to explain a concept may be omitted. Example on how to represent a `Load` event: `Load { address: 1000, value: I16(300) }`

Figure 4.3 shows an example host code on the top, that interacts with the example WebAssembly module shown in Listing 4.1. On the bottom the generated trace is depicted.

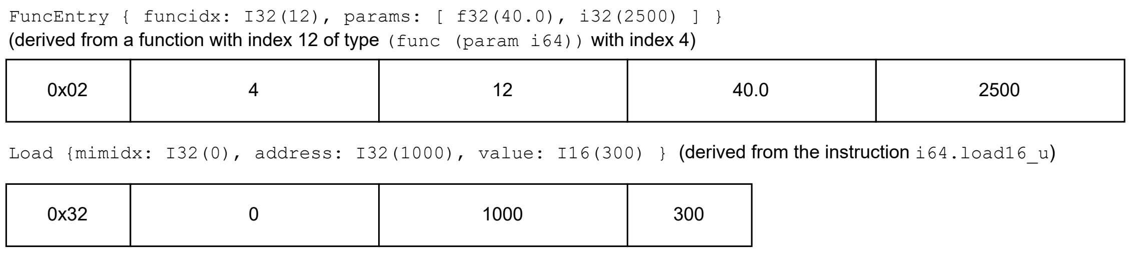
```
let imports = {
    env: {
        foo: () => 2
    }
}
WebAssembly.instantiate(getBinary/(), imports)
    .then(wasm => wasm.instance.exports.entry(1))
```

```
FuncEntry { funcidx: 1, params: [1] }
Load { memidx: 0, address: 1, value: 0}
Call { funcidx: 0 }
CallReturn { funcidx: 0, results: [2] }
```

Figure 4.3: Example trace generation

4.2.5 Trace Implementation

Wasm-R3's custom instrumentation as listed in Table 4.1, implementation uses a binary format as the in-memory and on-disk representation for storing the trace in a concise and easy to parse manner. Each entry starts with a byte that indicates the type of event that is represented. With the exception of the function entry and the call return event this byte also indicates the total byte length of the entry, and can be used to correctly parse the trace for processing. Since the number of parameters or results can differ between different function entries and call returns the next four byte word after the classifying byte for these events encodes a reference to the type of the respective function. During decoding this word can in combination with the original WebAssembly be used to figure out the internal structure of the trace entry. After these indicating bytes the actual information about the event is stored. Figure 4.4 shows the binary encoding of two trace events.



```
FuncEntry { funcidx: I32(12), params: [ f32(40.0), i32(2500) ] }
```
(derived from a function with index 12 of type `(func (param i64))` with index 4)

0x02	4	12	40.0	2500

```
Load {mimidx: I32(0), address: I32(1000), value: I16(300) }
```
(derived from the instruction `i64.load16_u`)

0x32	0	1000	300

Figure 4.4: Two examples for concrete trace entries that are encoded in the binary trace format.

To enable the trace to be read and edited by humans, there is a textual representation of the WebAssembly binary format. This is an intermediate form designed to be exposed in text editors or the console. In the textual representation entries are separated by a newline symbol `\n`. Each line starts with an indicator string followed by the event related information. Even though the textual representation is mostly designed for humans it can makes sense that programs use it internally for simplicity reason. In fact, the concrete Wasabi based implementation of Wasm-R3 relies only on the textual trace form.

4.2.6 Instrumentation Implementation

In Section 2.3 we layed out the differences between intrusive and non-intrusive instrumentation. In a web application context the specific meaning of intrusive and non-intrusive need to be defined more precisely. Since we regard only the WebAssembly module as the application and the JavaScript host code as the environment, we define adding additional instructions to the JavaScript hostcode as non-intrusive. To collect information about the WebAssembly execution we decide to use an intrusive approach, since this allows us to potentially perform the record in a more performant way and in addition makes the recording engine platform independent. This enables us to easily extend Wasm-R3 to record applications that are not only run on Chromium but also in other browsers or standalone runtimes such as NodeJs or Wasmtime. The intrusive collection works by extending the original code of the WebAssembly module with additional instructions or even entities such as additional functions or memories to collect and preserve the internal state of the program at specific points in time.

Wasm-R3 would potentially also be possible by instrumenting only the JavaScript code, however we decided for a WebAssembly only instrumentation, to keep the recording logic simpler.

The decision for an intrusive approach leaves us with two options: A) Using a third-party library or framework to perform the instrumentation or B) conducting a custom instrumentation. We decide to implement both.

As third-party instrumentation technology we use Wasabi, a framework for dynamically analyzing WebAssembly [25]. This technology works by calling into hooks at certain events during the WebAssembly execution. These hooks are JavaScript functions defined by the user that provide information about the occurred event as parameters. To make Wasabi runnable in the browser during record we compile it to WebAssembly.

An advantage using Wasabi is that writing the recording logic is simple and high level which makes it scalable and easy to reason about. A disadvantage is the performance overhead of crossing the WebAssembly to JavaScript context. It is also not possible to skip the instrumentation of certain events such as function entries of a non imported function which disallows many optimizations. Using Wasabi we face major performance bottlenecks that make the recording of many computing intensive web applications impractical.

Our custom instrumentation is, such as Wasabi, written in Rust and compiled to WebAssembly. During instrumentation it performs two passes over the WebAssembly module. The first pass collects information about the indices of entities, while the second adds the instrumentation instructions. To collect and store the trace, a secondary memory gets added along with a trace pointer global, initialized to 0 that points at the next free memory location to store the next trace event.[1] The instrumenter adds instructions in all relevant positions to collect runtime information, such as each function entry and load instruction. Additionally it adds utility locals to each function that hold critical values to prevent them from being consumed to early. The inserted instruction sequence always follows a common pattern.

As an example we will consider the instrumentation of a `i32.load` instruction in Listing 4.3. The custom instrumentation keeps the trace events as the in Section 4.2.4 introduced binary format in memory. This format starts with a indicator byte which represents the type of event. The first three instructions in Listing 4.3 are responsible for storing this indicator byte to the trace memory. In case of an `i32.load` instruction its value is hexadecimal 28. In line 6 the actual instrumentation logic starts. A load instruction pops one value from the operand stack, which is the load address. It also pushes one value to the operand stack, which is the value actually loaded. Both of these values need to be contained in the trace event. In order to prevent the address value from being consumed, it first gets saved in an auxiliary local, followed by the original load in line 7. Like the address the loaded value also gets saved in a auxiliary local. Lines 9-11 are responsible for first

[1]This approach is only applicable when the engine that hosts the recording implements the multi-memory proposal of WebAssembly

Listing 4.3: Custom instrumentation of a i32.load instruction

```
1     ;; store event type to trace
2     global.get $trace_pointer
3     i32.const 0x28
4     i32.store8 $trace_mem
5     ;; load instruction and its instrumentation
6     local.tee $addr
7     i32.load
8     local.tee $i32
9     global.get $trace_pointer
10    local.get $addr
11    i32.store $trace_mem offset=1
12    global.get $trace_pointer
13    local.get $i32
14    i32.store $trace_mem offset=5
15    ;; increment trace pointer
16    global.get $trace_pointer
17    i32.const 9
18    i32.add
19    global.set $trace_pointer
```

appending the address to the trace event. It gets stored at the location of the trace pointer plus an offset of 1 since the first byte is already populated by the event type. Lines 12-14 store the loaded value following the address in the same manner. This concludes the storing of the trace event and the trace pointer is ready to point to the next free address in the trace memory. Lines 16-19 perform the incrementation of the trace pointer. The value 9 in line 17 is the length of the previously stored trace event.

4.2.7 Trace Transfer Implementation

Wasm-R3 executes code in two distinct contexts. As shown in Figure 4.2 one context is a server which initiates the recording and takes the generated traces for replay generation, while the other context is the browser, which runs the web application under record together with the recording runtime. Due to security considerations processes running in the browser are not allowed to use the system interfaces of the operating system to directly store the trace to the filesystem, where it could be picked up by the server thread. For the server to obtain the traces, they have to be sent via technologies such as http.

There are two concrete approaches we applied to transfer traces from the browser to the server as shown in Table 4.1. A simple technique is bulk trace transfer. When applying this technique for the duration of record the trace gets collected and kept in-memory. On stopping the recording, this collected data gets sent in one go to the server, which starts consuming it for replay generation. While being simple to implement this approach has its limitations. Traces can grow up to millions of entries and thus can occupy multiple gigabytes of memory. When recording computing intensive applications, traces will typically grow larger then the available memory space. This happens

Listing 4.4: Monkey patchin of WebAssembly instantation function

```
const originalInstantiate = WebAssembly.instantiate
WebAssembly.instantiate = function(binary, imports) {
    binary = instrument(binary)
    imports = extendImports(imports)
    originalInstantiate(binary, imports)
}
```

quickly with the raw trace but also occurs when only keeping the optimized trace in memory. The problem can be partially solved by applying compression algorithms on the trace data structure, however this only works to a certain extend and damages the recording performance.

To prevent a process from running out of we replace bulk trace transfer by a streaming approach. Streaming the traces to the server requires some additional logic in the recording runtime. This logic needs to keep track of the trace size and if it passes a specific threshold trigger a send event to send it to the server. Afterwards the memory space occupied by the trace has to be freed again. This approach works very well with the custom instrumenter, but it can cause complications with the Wasabi based instrumentation since the trace resides as a JavaScript object. In JavaScript freeing of the memory is conducted by a garbage collector that will impose considerable performance overhead on the recording runtime, if the recording code is not written in a way that it is easy for the garbage collector to determine weither the memory occupied for the trace object can be freed or not.

4.2.8 Proxy Implementation

One central problem Wasm-R3 needs to solve is attaching the recording runtime to the website under record. Since it is a goal of Wasm-R3 to enable the recording of arbitrary web applications, the implementation commonly needs to deal with websites of unknown structure. Html is a very brought protocol, which makes reasoning about the production architecture of web applications difficult.

The most reliable way to intercept WebAssembly modules for instrumentation is through monkey patching the instantiation functions of the WebAssembly JavaScript Interface. We do that by overwriting functions such as `WebAssembly.instantiate` to accept the binary, add the instrumentation, and call the original function. This method is inspired by Wasabi [25]. An example of this monkey patching code is shown in Listing 4.4. Is is typically required to also add additional functions to the imports object since the instrumented code interacts with the host environment in a different way to exchange information.

Monkey patching in reality is a lot more complex. Some web applications call instantiation functions multiple times to run multiple WebAssembly modules simultaneously. The mechanism to support this behavior are complex and too space consuming to be explained in this thesis.

A further difficult challenge are different browser contexts. JavaScript itself is single threaded, however parallel processing is enabled through mechanisms such as workers and iframes. These contexts can only communicate between each other via specific protocols. As a result, attaching the recording runtime to only one of these contexts is not enough to capture any potential WebAssembly execution, instead the record runtime needs to be attached to all of the different contexts. To achieve this we choose to intercept every single file that gets sent to the browser as part of the website and statically determine if it is a JavaScript file. If it is, we add our idempotent runtime code to the beginning of the file. This solution works reasonably well, however it is not a universal solution since a lot of edge cases are not covered as for example inline JavaScript code that is either executed as a worker script or through eval. Up to this day, we were not able to solve this problem in a universally applicable way, which, as a consequence makes Wasm-R3 not applicable to many websites out there.

In addition to the described challenges above a whole set of different edge cases can occur that complicate the attachment of the recording runtime. One such case are html security headers. We solved this case by simply stripping security related headers of the main html file. There are still many unsolved challenges left.

4.3 Reduce

Wasm-R3 produces large traces. To keep memory and storage consumption on a reasonable level we optimize the trace by filtering out events that supply redundant information and are not needed for replay generation. Through reduction which is a form of optimization we transform the raw trace to the optimized trace. The replay generator will only need the optimized trace as an input. In Sections 4.3.1, and 4.3.2 we introduce two separate optimization approaches. Later in Section 4.3.3 we explain different architectural strategies to implement reduction in Wasm-R3.

4.3.1 Shadow Optimization

By tracing all instructions that read from the applications store, we pollute our trace with a lot of redundant events. Listing 4.5 shows an example.

Listing 4.5: Trace with redundant load events

```
1    Load { memidx: 0, addr: 0, value: 1 }
2    Load { memidx: 0, addr: 0, value: 1 }
3    Load { memidx: 0, addr: 0, value: 2 }
```

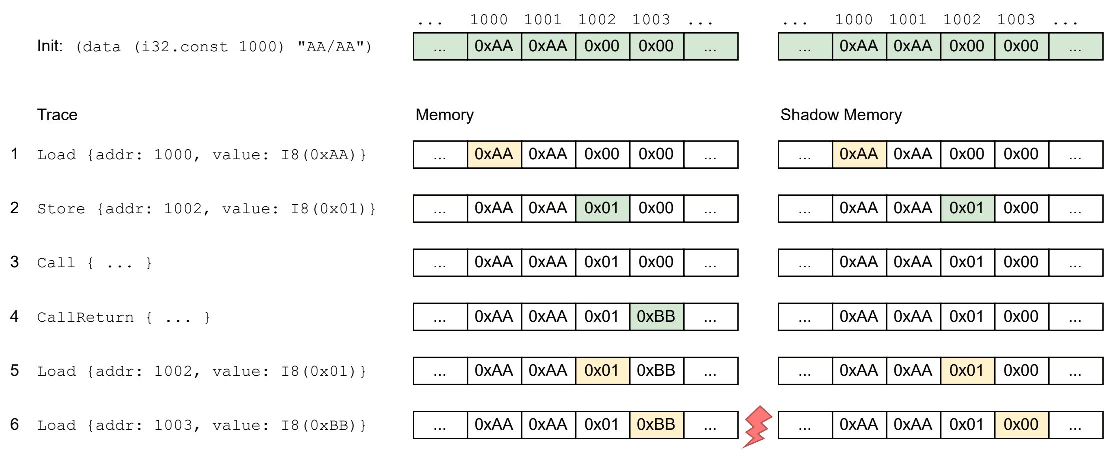
The load event at line 2 is redundant since it is already well known from the event at line 1 that the value of the memory at address 0 is 1. Similarly the event at line 3 is redundant since there have been no calls to function of the first order environment, and the change of the memory state happened due to WebAssembly's internal computation.

Our optimization needs to keep track of these internal influences and detect trace events that are redundant. Only if information that was created or changed by the host environment E is read, a corresponding event needs to be part of the trace. We call this class of optimizations shadow optimizations. As the part of the WebAssembly store that can be public can consist out of memories, tables and globals, we call the optimization performed on these entities shadow memory optimization, shadow table optimization and shadow global optimization respectively. The underlying concepts will be explained by using shadow memory optimization, however the overall strategies apply for all.

To perform the reduction of load instructions we add a new data structure that mimics the actual corresponding memory in the applications store, but is not mutated by the host environment. This data structure is called shadow memory. The algorithm is as follows

1. Initialize the shadow memory
2. On every store instruction, such as `i32.store`, update the shadow memory in the same way as the actual memory
3. On every load instruction check if the value loaded from the shadow memory differs from the value loaded from the actual memory
4. If the value is not equal keep the event
5. If the value is equal discard the event

To initialize the shadow memory the data sections of the WebAssembly binary are used, the initial memory content that is not explicitly defined is assumed to be zero. If the host environment initializes the memory as well, this information will still be encoded within the trace. Figure 4.5 illustrates the optimization on an exemplary trace. To make this work we extend the trace definition of Listing 4.2 by an additional event that holds information about store instructions: `Store := { memidx: I32, address: I32, value: ValType | I8 | I16 }`. We omit the field `memidx` and assume implicitly `0` to keep the example concise.



Init: (data (i32.const 1000) "AA/AA")
... 1000 1001 1002 1003 ...
... 0xAA 0xAA 0x00 0x00 ...
Trace
Memory
Shadow Memory
1 Load {addr: 1000, value: I8(0xAA)}
... 0xAA 0xAA 0x00 0x00 ...
2 Store {addr: 1002, value: I8(0x01)}
... 0xAA 0xAA 0x01 0x00 ...
3 Call { ... }
4 CallReturn { ... }
... 0xAA 0xAA 0x01 0xBB ...
5 Load {addr: 1002, value: I8(0x01)}
6 Load {addr: 1003, value: I8(0xBB)}

Figure 4.5: Shadow Memory Optimization

The example shows an initial memory configuration of the hexadecimal values `0xAA` and `0xAA` at addresses 1000 and 1001. Depicted in column two is the current state of the stores actual memory during record, while column two shows the shadow memory. If a certain value gets written at a specific step, the corresponding cell is colored green. If a certain value gets read, the corresponding cell is colored yellow. For every load both read values get compared with each other. A mismatch is indicated by a red lightning. Trace event 1 indicates that a value of hexadecimal `0xAA` has been loaded from address `1000`. Since the values in both memories match this event can be safely discarded. Event 2 indicates that hexadecimal `0x01` has been stored at address `1002`. This event updates both, the actual memory and the shadow memory. Since the store event is not needed for replay generation it can be discarded. Events 3 and 4 indicate that the WebAssembly process called a host function. After the host function the state of the actual memory has changed. The byte at address `1003` now holds the value `0xBB`. Event 5 indicates a load from address `1002`, similarly to event 2 actual memory and shadow memory match so we discard it. Event 6 loads the value `0xBB` from address `1003`. Actual memory and shadow memory do not match, which means this event needs to be kept in the trace. After event 6 the byte at address `1003` of the shadow memory needs to be updated to match the actual memory.

This algorithm is more complicated in reality since the matching needs to be performed not only for instructions such as `i32.load8_u` but also for instructions that affect multiple bits, which imposes additional challenges in the comparison of values. Section 4.3.3 will go over a concrete implementation of this reduction as on-the-fly optimization which does not require an extra `Store` event. In chapter 5.3 we will evaluate the effectiveness of the shadow optimization.

4.3.2 Call Stack Optimization

Shadow optimization filters out most of the execution store's related events. There are events related to function calls such as FuncEntry and CallReturn that are not affected by it. However, many events in this category are redundant as well. Listing 4.6 shows such an example trace. As WebAssembly module, we assume a module without any imported functions.

Listing 4.6: Trace with redundant call and function entry events

```
    Call { funcidx: 0}
    FuncEntry { funcidx: 0, params: [] }
    FuncReturn { funcidx: 0 }
    CallReturn { funcidx: 0, results: [] }
```

All four events in the depicted trace are redundant since a internal function gets called and no nondeterminism can be introduced. In fact most `call` instructions call WebAssembly internal functions. We can further conclude that most function entries happen due to an internal calls. Since these events do not involve any host interaction we can safely discard them.

As a central data structure this optimization uses a call stack. The elements of this call stack are an enum with the variants `Internal` or `External`. The algorithm is as follows:

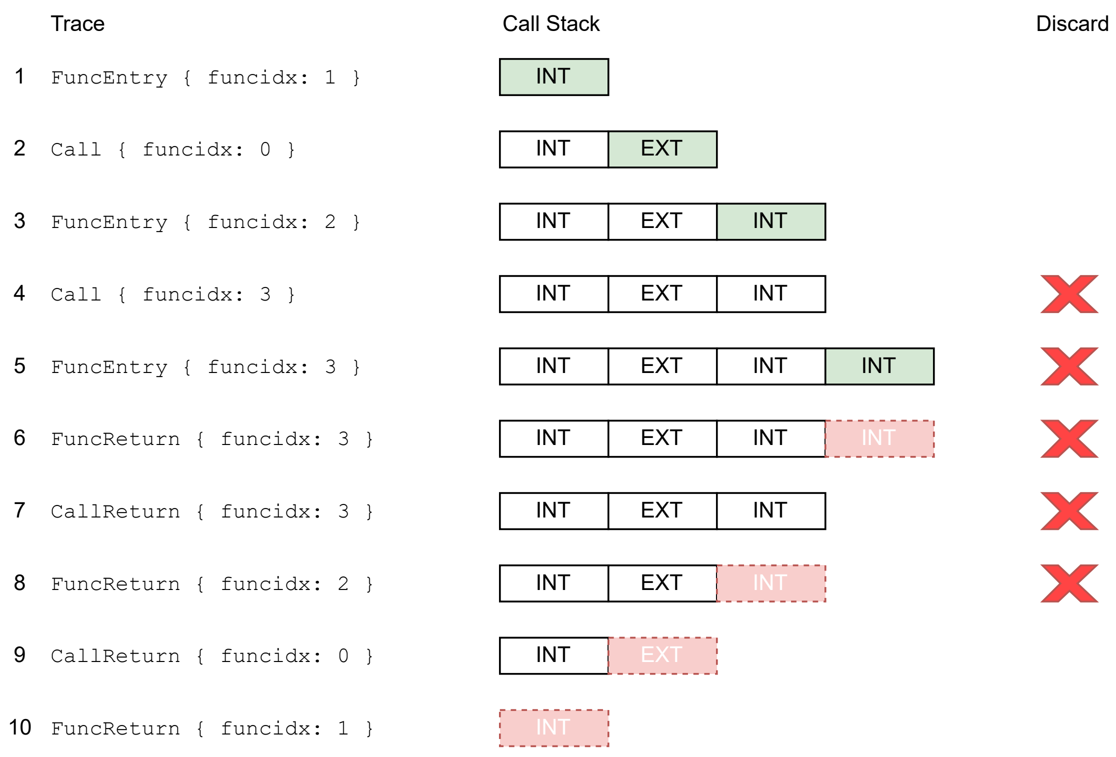
1. On every function entry push `Internal` on the stack. If the stack is either empty or the top has been `External` keep the event
2. On every call check if the function index of the callee is of an imported function. If yes push `External` on the stack and keep the event such as `i32.store`, update the shadow memory

in the same way as the actual memory

3. After every call check if the top of stack is of variant `Internal`. If no pop and keep the event
4. On every function return pop. If the stack is empty keep the event

Figure 4.6 illustrates this optimization by an example. Events that can be discarded are marked with a red cross.

```
(module
    (import "env" "foo" (func (;0;)))
    (func (;1;) (export "entry")
        call 0
    )
    (func (;2;) (export "bar")
        call 3
    )
    (func (;3;))
)
```



```
1  FuncEntry { funcidx: 1 }
2  Call { funcidx: 0 }
3  FuncEntry { funcidx: 2 }
4  Call { funcidx: 3 }
5  FuncEntry { funcidx: 3 }
6  FuncReturn { funcidx: 3 }
7  CallReturn { funcidx: 3 }
8  FuncReturn { funcidx: 2 }
9  CallReturn { funcidx: 0 }
10 FuncReturn { funcidx: 1 }
```

Figure 4.6: Call Stack Optimization

Event 1 indicates that function `1` has been called. Since the call stack is empty the event needs to be kept. `Internal` is pushed onto the stack. In event 2 function `1` calls imported function 0. `External` is pushed onto the stack and the event is kept. Event 3 again indicates a function entry. Since the top most element on the stack is `External` the event needs to be kept. Event 4 is a call to an internal function. The event does not need to be kept, nothing needs to be pushed onto the call stack. The function entry in event 5 does not need to be kept since `Internal` is on top of the stack. Another `Internal` gets pushed. In event 6 the first function return is encountered and the top most element gets popped from the stack. As long as the stack is not empty, this event can always be discarded. When discovering the first call return in event 7, `Internal` is on top of the stack. The event can be discarded but nothing gets popped. After another function return event 9 is reached. In this call return `External` is on top, which gets popped, while the event needs to be kept. The last event 10 indicates the return of the entry function. The top element from the stack gets popped which leaves an empty stack. As a consequence the event needs to be kept.

This algorithm has great optimization potential in WebAssembly applications where most function calls do not cross the WebAssembly to JavaScript boundary. In chapter 5.3 the effectiveness of the *call stack optimization* is evaluated.

4.3.3 Implementation

Considering the architecture depicted in figure 4.2 the reduction step can take place at multiple steps in the pipeline. Reduction can happen synchronously during record, which means that only the reduced trace is recorded and kept in memory. We refer to this strategy as on-the-fly optimization. This approach has the advantage of consuming only minimal amounts of memory but may in certain scenarios add additional logic to the recorder that can lead to a performance overhead. This is however not always the case.

In contrast to on-the-fly optimization reduction can also happen after the trace reached the server and has potentially been stored to the file system. We call this approach asynchronous optimization. This strategy keeps the complexity of the recorder logic low, might however lead to large traces that can be difficult to process. A common problem that occurs with this strategy is that the recording process runs out of memory and crashes. This will happen early if the process keeps the full trace in memory at once but also happens due to back pressure since the trace might grow faster then a possible stream is able to consume. Experience shows that many real world applications cannot be recorded with reasonable performance by using asynchronous optimization. As a third option it is possible to mix the above described approaches and perform parts of the optimization on-the-fly and other parts asynchronously.

During the workflow of Wasm-R3 reduction will always take place at some step of the pipeline. At a latest instance the backend will perform this step by simply skipping all events that are not needed for replay generation.

4.4 Replay

The trace and the original WebAssembly binary together can be used to generate the replay, by only using a single pass over the trace. The replay however is generated in undefined ordering, which can be problematic for large replay binaries since they may occupy a lot of memory during generation. A replay gets generated by a replay generator which produces a replay intermediate representation or replay IR, described in Section 4.4.1. Section 4.4.3 describes optimizations that can be performed on the replay IR to reduce its size or runtime overhead. Section 4.4.4 demonstrates the capability of the replay IR to be output to multiple target languages and output formats. Wasm-R3 uses the in Section 2.4 introduced replacing or ignoring strategies to mimic the environment. This ensures high portability of the generated benchmarks.

4.4.1 Replay Intermediate Representation

A replay needs to complement the original WebAssembly binary. It needs to provide all imported entities as well as the host actions to reproduce the program behavior during record. Listing 4.7 shows a simplified replay IR definition that ignores public tables and globals and mutable imported memories, i.e. we assume the public part of the store to consist out of only one exported memory. We simplify further and support only one byte stores at a time, which does not alter the replays semantics, but limits us in our application of replay optimizations. A tuple is denoted by brackets.

Listing 4.7: Definition of Replay Intermediate Representation

```
Replay      := { actions: Vec<Action>
                 funcs: Map<ImpDesc, Func>,
                 memories: Map<ImpDesc, Memory>}
Func        := Vec<Context>,
Context     := (Vec<Action>, Vec<ValType>)
Action      := ExportCall | MutateMem
ExportCall  := (String, Vec<ValType>)
MutateMem   := (String, I32, I8)
Memory      := Vec<I8>
ImpDesc     := (String, String)
```

A replay is defined as an object whose fields correspond to each of the original WebAssembly modules import types. Since a WebAssembly module can import multiple entities of the same type an entity gets represented by a map, that maps import descriptions to the actual entity definitions. An import description mirrors the import in the official WebAssembly specification as it consists out of a module and a name, both represented as strings. A memory is just a vector of `I8` values. A function is more complex since it contains the actual actions the host environment performed during record. Next to the imported entities the replay owns a field actions. This is the sequence of actions that were performed by the host environment independent of an imported function call. This includes the initial call of one of the WebAssembly modules exported functions as well as initialization of the stores public entities.

To make understanding of the replay IR more intuitive, we provide an encapsulated minimal example of an replay IR and how it would translate to the concrete representation in JavaScript in figure 4.7.

As the figure shows, the replay IR defines how to construct the WebAssembly import object as well as how to interact with the instantiated module. In line 26 the JavaScript replay code first instantiates the original module, with the custom import object. In line 27 the export object of this instance gets set to a global variable. These two lines are boilerplate code that are contained in any JavaScript replay and are not affected by the replay IR. Starting from line 26 the interaction with the module starts. We refer to that section as *global context*. These lines of code are directly mapped to the replay IRs actions field. Line 28 and 29 initialize the exported memory to the state that was observed during record. Line 30 is a call to the modules function exported as `main`. The import object is constructed through the other fields of the replay IR.

The example does not contain any imported memories, but the two functions `foo` and `bar`. Functions are a vector of context tuples. A functions has one context for each time the function was called during record. That way it is possible to omit the recording of input parameters. In the JavaScript the context vector get transformed into a switch statement, where each element directly maps to one switch case. A function bound static counter keeps track of the current count of calls of that specific function and helps to enter the correct context which provides the appropriate actions. Inside of each case, context actions get mapped in the same way as in the global context, to then be terminated by a return whose values are provided by the contexts second tuple element.

4.4.2 Non Monotone Replay Generation

The replay IR gets generated by a replay generator that takes as an input the original WebAssembly module and the optimized trace. The trace can be read succeedingly, typically by calling a function `consume_event` and each event will either add information to the replay or change the internal state of the generator. From the generators perspective the trace holds two kinds of events. (1) Action events, which are mapped directly to replay actions such as load, which maps to the action `MutateMem` and function entry, which maps to the action `ExportCall`, and (2) structural events, that provide information about the replays structure, i.e. where to insert actions in the replay, such as call or call return. This distinguishing between action events and structural events, as well as the internal state of the generator are necessary due to decoupled recording introduced in Section 4.2.6. This optimization states that instead of recording the whole store at every environment interaction, only the instructions related to the store are recorded. This introduces the challenge of what we call non monotone replay generation.

On a theoretical level we are able to record an application without applying any optimizations to neither the recording process nor the trace itself. In this simple scenario we would record the parameters and the state of the store on each function entry as well as the results and the state of the store at each call return. If we imagine a sequence of actions in a replay each trace entry in a trace such as this would end up in the generation of one or multiple actions that will get added at the end of the replay sequence. We call this process monotone replay generation. Figure 4.8 exemplify this concept. Writing an generation algorithm for such a replay is simple, however recording traces in that unoptimized form might be impractical and thus was not considered for Wasm-R3.

In contrast to monotone replay generation there is non monotone replay generation. During record store changes are observed at positions that are decoupled from the actual occurrence in the programs execution. This means that observed store changes have to be not only reordered within the current action sequence but also distributed amongst the different generated functions.

A trace that illustrates the challenge of non monotone replay generation is depicted in figure 4.8.

Listing 4.8: Trace recorded through decoupled recording

```
1 FuncEntry { funcidx: "main" }
2 Call { funcidx: "foo" }
3 FuncEntry { funcidx: "bar" }
4 Call { funcidx: "baz" }
5 CallReturn { funcidx: "baz" }
6 FuncReturn { funcidx: "bar" }
7 CallReturn { funcidx: "foo" }
8 Load { address: 0, 1}
9 Load { address: 1, 1}
```

As depicted in lines 8 and 9, two load events are observed that happened in the host environment at some point in time before the observation. It could have happened before the first function entry observed in line 1. It could also have happened in the imported function `foo`. If it happened in function foo, it could have happened before this function again calls the from the WebAssembly module exported function `baz` or after.

Our solution to this problem is a complex algorithm, whose details are too sophisticated to be explained in this thesis. The algorithm guarantees that the observed events during recording are observed in the same order during replay. It does however not guarantee that e replay actions are happening at the exact same spot as they where happening during record.

4.4.3 Optimizations

An advantage of generating the replay as an intermediate representation is that it is possible to perform optimizations on that IR that do apply for all possible output formats. Optimizations are not only effective to limit the runtime of the replay to a minimum, but might even be crucial for some engines to run the replay at all. V8 as an example disallows function sizes above a certain threshold which becomes problematic when a certain replay function holds a large number of contexts and actions. One possible optimization that could be performed here is the detection of repetitions and other patterns on different contexts consolidate them into a smaller reusable unit. Another strategy could simply split up generated functions into multiple smaller functions to limit the absolute size. To implement optimizations effectively the replay IR definition of Listing 4.7 needs to be extended. The overall structure could be more complex and new action types could be added. There could for example exist multiple `MutateMem` actions for different bit sizes, which a backend could use to output the appropriate concrete instructions.

4.4.4 Backends and Output Formats

Through a backend the replay IR can be output in different formats. Wasm-R3 implements backends for JavaScript replays and standalone WebAssembly backends. This design enables the replay on a wide variety of platforms. Javascript runtimes such as Node and Deno for example can run

WebAssembly only if it is instantiated via browser APIs such as `WebAssembly.instantiate` inside of a JavaScript file. Other engines such as Wizard [35] or Wasmtime might not support JavaScript but instead run self contained WebAssembly modules by executing their `entry` function.

```
memories: Map {},
actions: [
  MutateMem ("mem", 12, 123),
  MutateMem ("mem", 13, 123),
  ExportCall ("main", []),
]
funcs: Map {
  ("env", "foo"): [
    [[MutateMem ("mem", 14, 123), ExportCall ("add", [1, 1])], []],
    [[MutateMem ("mem", 15, 123)], []]
  ],
  ("env", "bar"): [
    [[ExportCall ("add", [2, 2])], [4]]
  ]
}
```

```
let exp
let fooCounter = -1
let barCounter = -1
const imports = { env: {
    foo: () => {
        fooCounter++
        switch (fooCounter) {
            case 0:
                exp.mem[14] = 123
                exp.add(1, 1)
                return undefined
            case 1:
                exp.mem[15] = 123
                return undefined
        }
    },
    bar: () => {
        barCounter++
        switch (barCounter) {
            case 0:
                exp.add(2, 2)
                return 4
        }
    }
}}
let wasm = await WebAssembly.instantiate(getWasmBin(), imports)
exp = wasm.instance.exports
exp.mem[12] = 123
exp.mem[13] = 123
exp.main()
```

Figure 4.7: Example replay IR and its translation to JavaScript

```
FuncEntry { funcidx: "foo", mem: [1, 1] }
FuncEntry { funcidx: "bar", mem: [1, 1] }
FuncEntry { funcidx: "foo", mem: [2, 2] }
```

```
MutateMem ("mem", 0, 1),
MutateMem ("mem", 1, 1),
ExportCall ("foo", [])
ExportCall ("bar", [])
MutateMem ("mem", 0, 2),
MutateMem ("mem", 1, 2),
ExportCall ("foo", [])
```

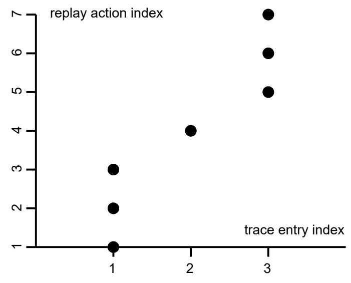


Figure 4.8: **Monotone Replay Generation.** Above: Recorded trace, **Left:** Generated replay actions, **Right:** Mapping from trace entry index to replay action index

5 Evaluation

We evaluate Wasm-R3 on the following 5 research questions.

- **RQ 1** How effective is Wasm-R3 in recording real-world applications?
- **RQ 2** How effective is the reduction of the trace?
- **RQ 3** Are the generated replays portable across a wide variety of WebAssembly engines?
- **RQ 4** Are generated replays accurately representing the original program during record?
- **RQ 5** How is the runtime performance of Wasm-R3

The evaluation is conducted using recording enabled through the Wasabi framework with bulk trace transfer. However the final integration of the custom tracer is expected to lead to significant improvements in the evaluation results for **RQ 1** and **RQ 5**.

5.1 Experimental Setup

The evaluation is conducted using the stable integrated Wasabi tracer, with bulk trace transfer. It is expected that measured values in recording performance and applicability will improve with the future integration of the custom tracer. This will also enable us to use larger evaluation sets.

We evaluate Wasm-R3 based on a set of 17 manually collected real world websites listed on *madewithwebassembly.com*. We selected listed apps based on two criteria: (1) The listed application is a website which runs WebAssembly and (2) wasm-r3 has the capabilities to instrument and record it. As described in Section 4.2.8 websites can be structured in a complex hard comprehensible way, which makes implementing a program that faithfully instruments all WebAssembly web applications a challenging task that cannot be completely resolved in the scope of this work. Another complication are the performance shortcomings of the Wasabi tracer with bulk memory transfer. Sometimes instrumentation works as expected but transferring the traces to the server works unreliably due to the process running out of memory.

As a result we limit our evaluation set to a total number of 17. As 5.2 will show, Wasm-R3 still works on a broader range of actual real world applications compared to the similar tool WasmView. The set represents a broad range of real world web applications, since the domains of the collected applications are highly variant. We categorized these domains in ten categories that are gaming, media editing, video players, programming language execution, animations, benchmarks, algorithms, Geography and utilities. These websites also differ significantly from each other in the size of traces they generate. When interacting with these websites through a simple workflow it becomes clear that some applications execute more instructions and generate raw traces that contain 100 or 1000 times more events then traces generated by a different application in the same time. Our test case *video* does instantiate a WebAssembly module, but does not call any of its imported functions, so it generates the empty trace. Table 5.1 lists the individual web pages of our collection and their approximate magnitudes in raw trace sizes through simple interaction.

Name	**Domain**	**Application**	**Events**
boa	PL Executor	`https://boajs.dev/boa/playground/`	10^6
commanderkeen	Game	`https://www.jamesfmackenzie.com/chocolatekeen/`	10^7
ffmpeg	Media Edit	`https://w3reality.github.io/async-thread-worker/examples/wasm-ffmpeg/index.html'`	10^5
fib	Benchmark	`https://takahirox.github.io/WebAssembly-benchmark/tests/fib.html`	10^9
figma-startpage	Animation	`https://www.figma.com`	10^3
funky-kart	Game	`https://www.funkykarts.rocks/demo.html`	10^7
game-of-life	Game	`https://playgameoflife.com/`	10^3
guiicons	Media Editor	`https://playgameoflife.com/`	10^6
handy-tools	Utility	`https://handytools.xd-deng.com/`	10^5
jsc	PL Executor	`https://mbbill.github.io/JSC.js/demo/index.html`	10^7
kittygame	Game	`https://wasm4.org/play/kittygame`	10^7
pathfinding	Algorithm	`https://jacobdeichert.github.io/wasm-astar/`	10^7
riconpacker	Media Editor	`https://raylibtech.itch.io/riconpacker`	10^6
rtexviewer	Media Editor	`https://raylibtech.itch.io/rtexviewer`	10^5
sqlgui	PL Executor	`https://sql.js.org/examples/GUI/`	10^5
video	Media Editor	`https://d2jta7o2zej4pf.cloudfront.net/`	0
multiplyInt	Benchmark	`https://takahirox.github.io/WebAssembly-benchmark/tests/multiplyInt.html`	10

Table 5.1: Real world websites evaluation set

As additional evaluation set we created a suite of 63 micro test cases which is a comprehensible list that portraits possible WebAssembly application runtime structures that need to be captured by a record and replay system to create accurate replays. This suite was initially created by carful examination of the WebAssembly specification and translation of the relevant language constructs into the separate test cases. The suite was extended constantly to portrait further challenges that occurred during development. At the current stage it is impossible to claim that the suite satisfies all possible runtime structures that affect replay, we are however confident that most language constructs are considered. This hypothesis is further supported by the fact, that we do not run into correctness issues when executing Wasm-R3 on arbitrary supported websites.

All experiments are performed on a desktop pc with an 8-Core Intel Core i9 (3.6 GHz, 16 MB L3 cache) and 32 GB of RAM. The operating system is macOS Catalina 10.15.6 64-bit. To record our benchmarks we use a developer build of chromium 119.0.6045.9.

5.2 Effectiveness of Recording (RQ 1)

We use an extended set of real world WebAssembly web applications to evaluate the effectiveness of Wasm-R3 in properly instrumenting and recording real-world web applications. We create this set by collecting all links listed on *https://madewithwebassembly.com* and *https://github.com/mbasso/awesome-wasm* via web scraping, which leaves us with a list of 339 websites. As a next step we filter out 33 links that are not accessible and another 225 that only point to static webpages that are not interactive. This leaves us with 80 websites out of these, we pick all websites that contain the string `WebAssembly.instantiate` in at least one of their served JavaScript or html files. In total we end up with a set of 59 web applications.

We run the Wasm-R3 recorder on all of these and check if trace gets recorded inside of the browser context. If yes, we consider Wasm-R3 to be applicable on that specific application. Our experiment shows that Wasm-R3 is applicable on 41 websites, which translates to about 69.49% percent of the extended evaluation set.

To assess this result we also run the comparable tool WasmView once on the same applications and evaluate its applicability. WasmView is suitable for this comparison, because it similarly to Wasm-R3 takes a URL as input, redirects the user two the respective website and then dynamically collects runtime information of the underlying program while the user interacts with it [31]. Based on our research we could not find other publicly available tools that are based on a comparable work flow. WasmView generates output on only 14 websites or 23.73% of applications.

This is due to a variety of reasons. WasmView is for example not able to detect WebAssembly modules loaded in iframes, as well as other diverse bugs that make the tool crash or not accept specific URLs. Based on these values we can claim that Wasm-R3 is broader applicable on real-world WebAssembly applications than other comparable tools out there.

5.3 Effectiveness of Reduction (RQ 2)

To evaluate the effectiveness of our trace reduction we run Wasm-R3 on the 17 websites of our evaluation set and compare the raw trace with the reduced trace. To get an overall impression, we first sum up the number of events of all collected raw traces and calculate the percentage of how many events remain in the optimized trace. We call the ratio between raw event count and reduced event count remaining ratio. We find that only of 0.1% of all events remain in the optimized traces. While this number seems small, a more granular analysis is required since the considered web applications have very different trace sizes, as presented in Section 5.1. This means a very good remaining ratio in *fib* will overshadow weak remaining ratios in other applications. To get more precise insights we calculate the remaining ratios for all event types found in the traces separately for each application. Figure 5.1 shows the results.

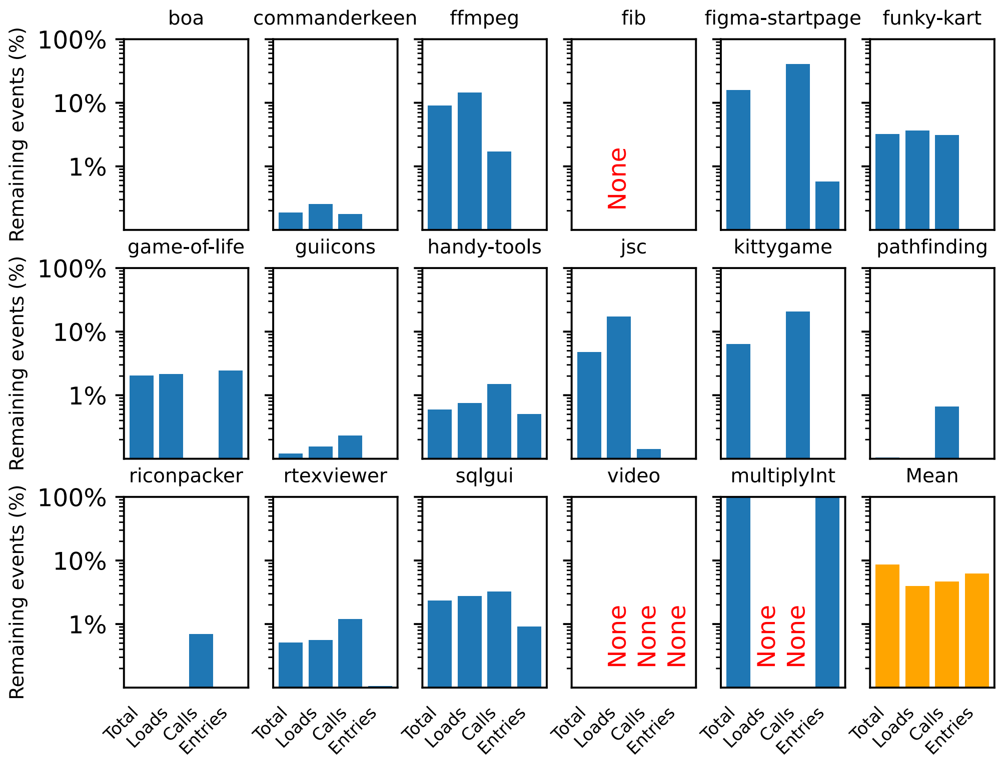


Figure 5.1: Remaining ratios for selected event types per application. Lower is better.

The x-axis lists the ratios for three different event types. Loads, function calls, and function entries. Loads evaluate our overall effectiveness for shadow memory optimization, while function calls and function returns indicate the effectiveness for the call stack optimization. Other call stack related event types are omitted, since they mirror results for function calls and function returns. As for table and global related events, we find that there remaining ratios are always 0%, which means that public tables or globals are never mutated by the host environment by the applications in our evaluation set. If this would be true for applications in general, this would enable significant complexity reduction and performance improvements for Wasm-R3.

The left most bar indicates the overall remaining ratio for this application. The y-axis indicates the remaining ratio in a logarithmic scale. Most of the time only between 0% and 1% of the raw trace events remain in the optimized trace. In only a few cases more then 10% of the events remain. The absence of any events of a certain type in an application is indicated by the red string *None*.

As seen in the first chart for the application *boa*, the remaining ratio is very low for all three event types. Precisely they are 0.02% for loads, 0% for calls and 0.005% for function entries. These values are even to small to be visible on a logarithmic scale. The bar chart for *multiplyInt* indicates a remaining ratio of 100% for function entries. These values however are not necessarily meaningful in a larger context since the overall raw trace of *multiplyInt* consists only out of 10 events. The three applications *figma-startpage*, *funkyKart*, and *kittygame* render animations. Applications of that type typically have a high remaining ratio for function calls, since they need to call into the environment multiple times per second in order to display the rendered frames.

The last bar chart indicates the means of the reduction ratios across all applications. These are 8.53% in total, 3.97% for loads, 4.65% for calls, and 6.17% for function entries. The standard deviations are 5.42% for loads, 10.83% for calls, and 24.14% for function entries respectively.

5.4 Portability of Wasm-R3 (RQ 3)

A goal of Wasm-R3's server architecture is the generation of replays in different output formats for them to be run on different engines. Currently JavaScript replays for runtimes such as NodeJs and self contained, standalone WebAssembly replays for engines such as Wizard or Wasmtime are supported. To evaluate the portability of our created benchmarks we first generate replays for the 17 web applications in our real world evaluation set and try to execute those on a variety of different engines. For standalone WebAssembly replays we use Wizard, Wasmtime, Wasmer, SM, V8 and JavaScriptCore. For evaluating the portability of JavaScript replays we use the well known runtimes nodejs, Deno and bun for execution. We run each of the replays once on each engine via a CLI invocation until completion. If a runtime reports an error we consider the benchmark not portable to this specific runtime. Table 5.2 shows the the the portability of created benchmarks to different WebAssembly engines.

Name	**Wizard**	**Wasmtime**	**Wasmer**	**SM**	**V8**	**jsc**	**node**	**deno**	**bun**
boa	✓	✓	✓	✓	✓	✓	✓	✓	✓
commanderkeen	✓	✓	✓	✓	✓	✓	✓	✓	
ffmpeg	✓	✓	✓	✓	✓	✓	✓	✓	✓
fib	✓	✓	✓	✓	✓	✓	✓	✓	✓
figma-startpage	✓	✓	✓	✓	✓	✓	✓	✓	✓
funky-kart	✓	✓	✓	✓	✓	✓	✓	✓	
game-of-life	✓	✓	✓	✓	✓	✓	✓	✓	✓
guiicons	✓	✓	✓	✓	✓	✓	✓	✓	
handy	✓	✓	✓	✓	✓	✓	✓	✓	
jsc	✓	✓	✓	✓	✓	✓	✓	✓	
kittygame	✓	✓	✓	✓	✓	✓	✓	✓	✓
pathfinding	✓	✓	✓	✓	✓	✓	✓	✓	✓
riconpacker	✓	✓	✓	✓	✓	✓	✓	✓	✓
rtexviewer	✓	✓	✓	✓	✓	✓	✓	✓	✓
sqlgui	✓	✓	✓	✓	✓	✓	✓	✓	✓
video	✓	✓	✓	✓	✓	✓	✓	✓	✓
multiplyInt	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 5.2: Portability of the replays generated for each application.

As depicted in the table, our benchmarks are portable through a wide variety of WebAssembly engines. Only the newly introduced JavaScript runtime Bun (v1.0.29) fails to execute 6 out of 17 replays.

5.5 Correctness of Wasm-R3 (RQ 4)

To validate that the by Wasm-R3 produced JavaScript replays do not modify the semantics of the original program, and indeed deterministically replay the same process as during record, hence to evaluate the overall correctness of the approach, we compare the behavior of the program during record with the behavior of the replay. For our test suite of 17 real-world applications we attach the recorder to the replay and generate traces once again. We then compare the optimized traces during record with the optimized traces captured during replay. If these two traces match, we consider Wasm-R3's behavior accurate. Similarly for our suite of micro tests we collect and compare the raw traces generated during record and replay. Our results show that Wasm-R3 is able to generate matching traces for all 17 real world programs as well as all 63 micro test cases. This result lets us conclude back on the overall correctness of the replay IR since it reflects the underlying structure of the concrete JavaScript representation. To validate that the by Wasm-R3 procured standalone WebAssembly replays are correct we generate them through the same test set. To ensure determinism we run the replays five times and ensure that they yield the exact same traces on every execution. and check if it is a valid WebAssembly module by verifying it through `wat2wasm`. [1] Since we know about the correctness of the replay IR we can assume that the generated WebAssembly replay also is accurate, however we cannot make any guarantees about the absence of bugs in the responsible backend.

We make our results reproducible by writing a script that that automatically runs Wasm-R3 on all websites and micro test cases. For each of these tests our script generates traces and replays and keeps them available after execution for potential debugging. In addition a report file is created that contains additional debugging information such as the full traces in case of a mismatch.

5.6 Performance of Wasm-R3 (RQ 4)

We collect several distinct metrics to evaluate the performance of Wasm-R3. In Section 5.6.1 we measure as an overview the wall time from start to end of a Wasm-R3 invocation. In Section 5.6.2 we measure the the recording overhead. Section 5.6.3 provides data about the replay generator performance. We measure the performance of executing the generated benchmarks in Section 5.6.4

5.6.1 Wall Time

To get an overview over the total performance of Wasm-R3 we measure the wall time. Using the `performance` object of the web's performance api, we measure the time from the initial launch of the recording of a web application until the generation of all related replays is finished. This is includes the startup and shutdown of the browser instance, the time it takes to transfer generated trace data and the original WebAssembly binary to the server via bulk trace transfer and the generation of the replays. For the measurement we automatically run Wasm-R3 on the 17 applications in our evaluation set. The interaction with the web application during record is defined by a simple

[1]https://github.com/WebAssembly/wabt

playwright script. To reduce the effect of the recording script onto the wall time we subtract the recording duration from the total measured time. We do this because the playwright script that defines the interaction with the website, sometimes relies on hardcoded fixed delays to perform actions. Including the record in the execution time would distort the results.

To make our measurements more robust to external factors we run the measurement N times with N=5 and calculate the mean. Figure 5.2 shows our results. We measure a shortest wall time of 876 milliseconds for *multiplyInt* and a longest of 37.913 seconds for *jsc*. This long wall time of *jsc* is explained by the large size of the WebAssembly binary, which takes in each of our five measurement repetitions more then 10 seconds to be sent to the server. As a mean wall time we measure 8.651 seconds, the standard deviation is 9.396 seconds.

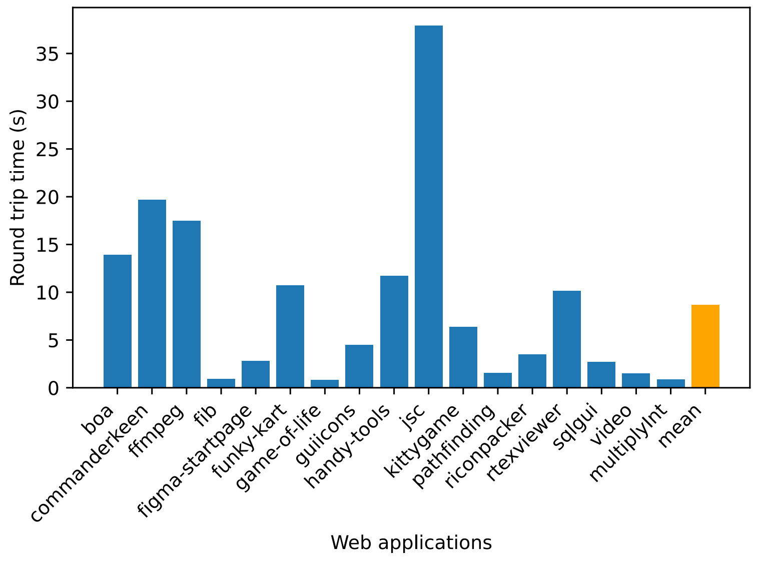


Figure 5.2: Wall time.

5.6.2 Recording Overhead

We measure the recording overhead of Wasm-R3 by selecting all web applications of our evaluation set where the playwright interaction script does not rely on static hardcoded delays. This is true for only five applications: *ffmpeg*, *fib*, *handy-tools*, *sqlgui*, and *multiplyInt*. Creating delay free playwright scripts for the other web applications is quite involved and could not be achieved in the limited working time of this thesis. We measure the time from the start of the run of the playwright script until its end. We again measure each execution N times with an N=5. Figure 5.3 shows the results.

As the chart shows, the recording overhead fluctuates quite a lot between a 6x slowdown for *ffmpeg* and not observable slowdown for *multiplyInt*, which results in a variance of 2.26. This shows that the overall slowdown factor is highly dependent on the internal structure of the application under record.

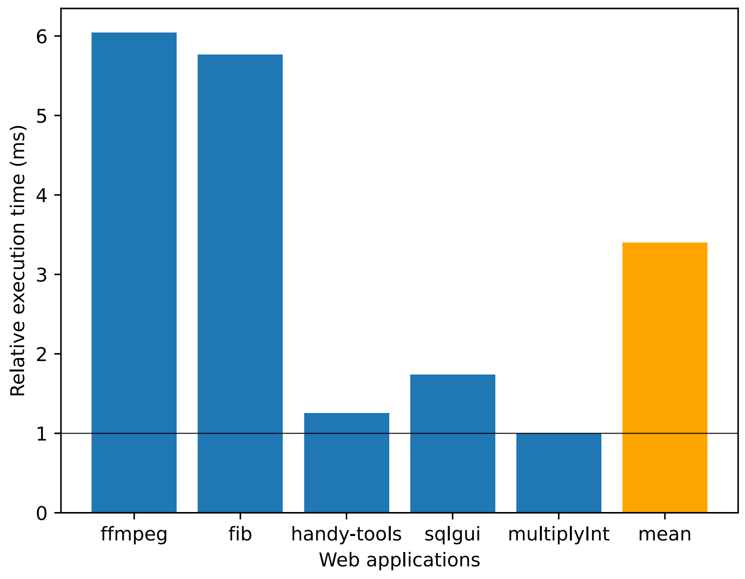


Figure 5.3: Relative recording of Wasm-R3 normalized to the uninstrumented website interaction

5.6.3 Replay Generation

We are interested in the time Wasm-R3 needs to generate replays. We run the same experiment as described in Section 5.6.1 but this time measure the time it takes for the replay generator to generate the replay IR and save it in the JavaScript representation to a file.

All of our test cases generate traces of different length, which as we expect will affect the time the replay generator runs. For this reason we divide the measured time by the number of events of the optimized trace and obtain the relative replay generation time. Figure 5.4 shows this relative replay generation time on a bar chart on the left. We omit the application *video* since it generates a trace with no events and thus makes a calculation of the relative replay time impossible. The shortest time we measure is 0.655 microseconds per trace event for the test *kittygame*, the longest time is 377.754 microseconds for the test *fib*. In the mean the relative replay time is 50.545 microseconds, the standard deviation is 99.666 microseconds. These numbers come as no surprise. *kittygame* has, since it needs to display the frames for its game, and thus has many calls to imported functions, large optimized traces. In contrast *fib* does not contain any store related events and only calls to internal functions which makes its optimized trace size very small.

We suspect that the relative replay generation time decreases with a larger trace sizes. To proof our suspect, we plot the relative replay generation time relative to optimized trace size in the right graphic of Figure 5.4. As shown in the plot we reach optimal relative replay generation time at optimized trace sizes higher then 100,000. The optimal relative replay generation time is lower than 1 microsecond. The increase in relative replay time for smaller trace sizes is explained by the startup time of the replay generator. Having a relative replay generation time lower than 1 microsecond means that the generation of a replay from a trace of event length 1,000,000 can be conducted in below 1 second added to the replay generator startup time.

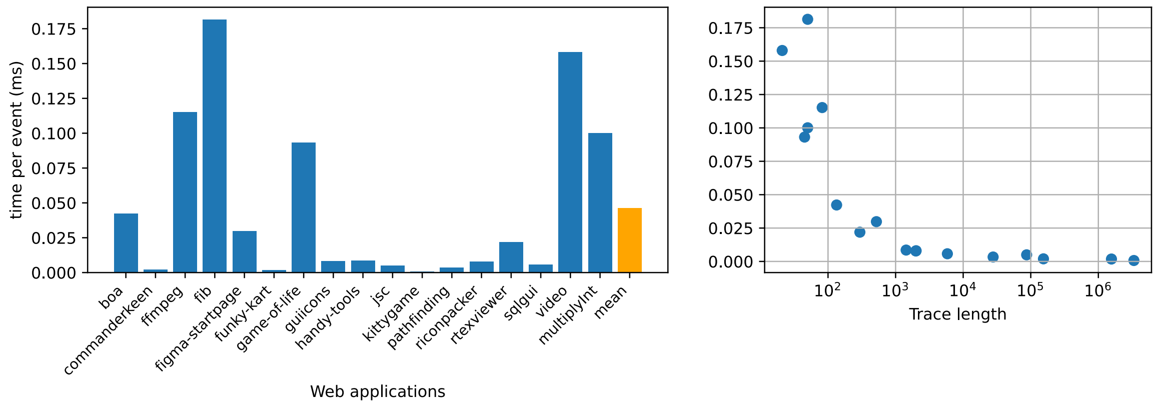


Figure 5.4: Relative Replay Generation time

5.6.4 Replay Execution Time

We measure the replay execution wall time, for our evaluation suite of 17 web applications. Again, we repeat the measurements N times with N=5 and calculate the mean. Figure 5.5 shows the results. With a wall time of above 3 seconds *fib* has the highest replay wall time. 11 out of 17 replays have a execution time below 100 milliseconds. The mean wall time is 633.00 milliseconds, the standard deviation is 987.60 milliseconds.

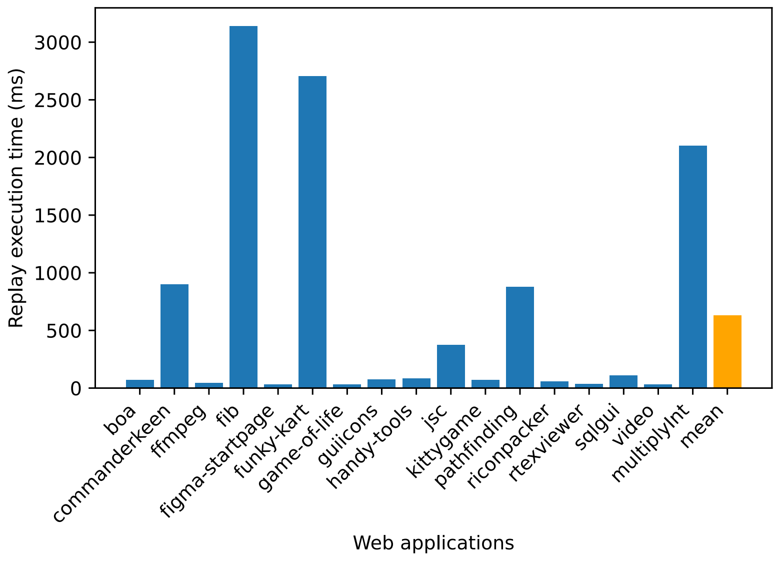


Figure 5.5: Replay execution wall time

6 Discussion

Wasm-R3 is still under active development. As to this day the implementation suffers from various different bugs and performance inefficiencies. Since the state of this tool is highly dynamic, data collected and presented in the evaluation section might be outdated soon in the future. The evaluation has been conducted on a stable but already outdated version of Wasm-R3.

Capturing the performance properties of Wasm-R3 is difficult. This is due to the complex multi step process a Wasm-R3 invocation triggers. Measuring the performance of recording differs significantly from the performance measurement of the replay generation. Another difficulty arises through the dynamic nature of Wasm-R3. Since the approach can and is implemented in many different ways, performance measurement results can differ significantly from one implementation to the other. Comparing the performance of different implementations between each other was not possible in the scope of this thesis.

Since Wasm-R3's proxy component struggles to support a wide variety of websites, we could only obtain a set of 17 real world applications to perform our evaluation. While being relatively versatile we expect to not have covered many use case scenarios of WebAssembly in our evaluation set. This has severe consequences especially on the evaluation of the correctness of Wasm-R3. The language constructs under consideration are collected manually in our micro test suite, which cannot be guaranteed to be complete. Wasm-R3 furthermore disregards some new instructions that are part of the WebAssembly specification 2.0.

It would be desirable to use a whole set of many other metrics to proof that our benchmarks have strong similarities to the original application. Our evaluation has shortcoming in that regard. We argue for the similarity by just comparing the generated trace during record with the generated trace during replay.

In Chapter 4 we introduced an abstract architecture for Wasm-R3. This architecture can be applied for web applications running in the browser, might however not be well suited for building record and replay systems for WebAssembly programs that run on the server.

7 Future Work

Wasm-R3 still houses a lot of potential for improvements. Currently the integration of the custom tracer is under active development. We expect the custom tracer to dramatically boost the recording performance, which will enable to Wasm-R3 to record a whole new class of applications such as hardware emulators and make the recording of other resource intensive applications such as games much more fluent. Due to not being limited to memory anymore, streaming the traces to the server during record will further allow to record larger traces and generate larger replays. To evaluate those improvements, an extensive evaluation of the recording performance needs to be conducted to show evidence, to show the approach's capabilities of generating benchmarks out of arbitrary applications.

Wasm-R3 uses playwright as a browser automation library to instrument the desired web applications and adding the recording runtime. Since playwright is developed for end-to-end testing of web applications it has shortcomings in its ability of analyzing and modifying live websites. Currently there are no uniformly working strategies available for tasks such as intercepting each WebAssembly module instantiated by a website, or adding runnable code to each browser context. Since the architecture of web applications is typically complex, we see a lot of use cases for tooling that simplifies the analysis and modifications of such applications and suggest that further research efforts should work on developing techniques to enable such tooling.

In the current state Wasm-R3 does not yet support all features of WebAssembly 2.0 such as for example bulk memory instructions. Ongoing efforts shall support these features and further implement current WebAssembly specification proposals to enable recording capabilities in the future. Furthermore we are planning to extend our program to not only be able to record web applications but also server side applications that get run at different runtimes such as NodeJs, Wasmtime and Wizard. This will improve the approach's ability to generate a broader range of different real world benchmarks.

A goal of our work is to provide a simple and quick solution to create benchmarks that are relevant and thus representative of real world applications. We expect to see engine developers to use our approach to improve the performance of their virtual machines by being able to reason more precisely about their implementations and approaches.

In the past record and replay techniques have typically been used for debugging purposes. The underlying concepts of record and replay stay the same independently of its final goal of creating a benchmark or reproducing the execution for debugging. Given that fact we expect that the principles presented in this thesis can be reused to create further development tools such as debuggers to help future WebAssembly developers to write more robust applications.

8 Related Work

Wasm-R3 is inspired by the work of Richards et al. who proposed a record and replay technique for the automated construction of JavaScript benchmarks [30]. They called the implementation of their technique JSBench. Since WebAssembly is a deterministic bytecode to which nondeterminism gets only introduced by its calls to imported functions, the overall approach of Wasm-R3 differs significantly from the underling approach of JSBench. Wasm-R3 does not rely on mock objects to record its traces, and collects information about nondeterminism in a non monotone manner. To record the WebAssembly execution traces need to be collected that typically grow a lot faster and exceed the total size of traces that get generated by JSBench. This introduces a whole class of optimization problems, that Wasm-R3 solves.

At the core of our instrumentation approach is Wasabi, a framework for dynamic WebAssembly analysis [25]. Wasabi works in a intrusive way by adding additional instructions to the WebAssembly binary and triggering a call to JavaScript functions on certain events. These functions are defined by the user and obtain all necessary information about the occurred event through arguments. Writing the recording logic for Wasm-R3 with the help of Wasabi is relatively simple and scalable. Through the WebAssembly to JavaScript call overhead however we receive significant performance penalties that prevent Wasm-R3 to record a whole bunch of web applications in reasonable time. Since a goal of our tool is to support arbitrary websites we are currently working on deprecating the Wasabi based recorder and generating traces through custom instrumentation.

Replay techniques have been extensively explored previously, primarily with a focus on debugging applications. Researchers such as Cornelis et al. [8] and Dionne et al. [12] have conducted surveys and classifications of replay-based debugging tools. According to Dionne's classification, Wasm-R3 stands out as a data-driven automatic replay system. It operates by capturing data exchanges between the program and its environment and does not necessitate manual modifications to the source code of the monitored program. A replay is considered unsuccessful if the program's interactions with its environment diverge from the recorded trace.

A variety of studies have investigated the runtime performance of WebAssembly, either compared to JavaScript [37, 9, 10, 36] or to native code [22, 23]. All of these rely on already existing benchmark suites such as PolyBenchC or Spec CPU which are not necessarily representative of real world applications. This shortcoming is a motivation for Wasm-R3 to better support future claims about WebAssembly's runtime performance. We also hope that benchmarks generated by our tool will speed up the optimization development of virtual machines.

Shadow memory is a concept widely used in Dynamic Binary Analysis (DBA) tools, which inspired our shadow optimization approaches. In [26], Nethermost et al. give a detailed description of robust shadow memory implementations. In [33], Sen et al. use shadow values to develop a selective record and replay framework for JavaScript.

9 Conclusion

This thesis introduced Wasm-R3 an approach for recording and replaying WebAssembly web applications and creating standalone benchmarks from them. We introduced the concept of environment E and application A to provide us with a basic framework to describe record and replay applications in general. We provided an extensive description of Wasm-R3's basic workflow via record, reduce, and replay and its abstract architecture. Furthermore we implemented the approach in Typescript and Rust and described our considerations and different approaches we took, as well as their advantages and disadvantages.

We evaluated Wasm-R3 and showed that our approach is able to successfully instrument around three times as many web applications as the already existing tool WasmView. We furthermore proofed the effectiveness of our developed trace reduction algorithms by executing our tool on a test suite of 17 real world applications and calculated that 99.9% of all recorded raw trace events can be filtered out. We further showed that all of these 17 generated benchmarks run on most popular WebAssembly engines.

By comparing the generated traces collected during record for our 17 real-world applications, with the traces generated during replay, we saw that Wasm-R3 is able to produce benchmarks that accurately represent the recorded application.

Lastly we showed that benchmarks can be generated in reasonable time, by measuring a mean wall time of 8.651 seconds.

Bibliography

[1] WebAssembly Specification. https://github.com/WebAssembly/spec. Retrieved Februar 18, 2024.

[2] WebAssembly website. https://webassembly.org/, 2024. Retrieved Februar 8, 2024.

[3] S. M. Blackburn et al. The dacapo benchmarks: Java benchmarking development and analysis. In *Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 169–190, 2006.

[4] Blindman67. Why is webassembly function almost 300 times slower than the same javascript function. https://stackoverflow.com/questions/48173979/why-is-webassembly-function-almost-300-time-slower-than-same-js-function, 2018. Retrieved Februar 9, 2024.

[5] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst. Interactive record/replay for web application debugging. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, UIST '13, page 473–484, New York, NY, USA, 2013. Association for Computing Machinery.

[6] W. Chen. Performance testing web assembly vs javascript. https://medium.com/samsung-internet-dev/performance-testing-web-assembly-vs-javascript-e07506fd5875, 2018. Retrieved Februar 9, 2024.

[7] ColinE. Why is my webassembly function slower than the javascript equivalent? https://stackoverflow.com/questions/46331830/why-is-my-webassembly-function-slower-than-the-javascript-equivalent/46500236#46500236, 2017. Retrieved Februar 9, 2024.

[8] F. Cornelis, A. Georges, M. Christiaens, M. Ronsse, T. Ghesquiere, and K. De Bosschere. A taxonomy of execution replay systems. 01 2003.

[9] J. De Macedo, R. Abreu, R. Pereira, and J. Saraiva. On the runtime and energy performance of webassembly: Is webassembly superior to javascript yet? In *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, pages 255–262. IEEE, 2021.

[10] J. De Macedo, R. Abreu, R. Pereira, and J. Saraiva. Webassembly versus javascript: Energy and runtime performance. In *2022 International Conference on ICT for Sustainability (ICT4S)*, pages 24–34. IEEE, 2022.

[11] S. Dieckmann and U. Hölzle. A study of the allocation behaviour of the specjvm98 java benchmarks. In *European Conference on Object Oriented Programming (ECOOP)*, pages 92–115, 1999.

[12] C. Dionne, M. Feeley, J. Desbiens, and A. Informatique. A taxonomy of distributed debuggers based on execution replay. 09 1996.

[13] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan. Repeatable reverse engineering with panda. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, PPREW-5, New York, NY, USA, 2015. Association for Computing Machinery.

[14] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review*, 36(SI):211–224, 2002.

[15] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, dec 2003.

[16] R. Fryer. Low and non-intrusive software instrumentation: a survey of requirements and methods. In *17th DASC. AIAA/IEEE/SAE. Digital Avionics Systems Conference. Proceedings (Cat. No.98CH36267)*, volume 1, pages C22/1–C22/8 vol.1, 1998.

[17] G. Gurgone and P. Spiess. A real-world webassembly benchmark, 2018.

[18] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the web up to speed with webassembly. *SIGPLAN Not.*, 52(6):185–200, jun 2017.

[19] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii. Chstone: A benchmark program suite for practical c-based high-level synthesis. In *2008 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1192–1195, 2008.

[20] A. Hilbig, D. Lehmann, and M. Pradel. An empirical study of real-world webassembly binaries: Security, languages, use cases. In *Proceedings of the Web Conference 2021*, WWW '21, page 2696–2708, New York, NY, USA, 2021. Association for Computing Machinery.

[21] A. Janes, X. Li, and V. Lenarduzzi. Open tracing tools: Overview and critical comparison, 2023.

[22] A. Jangda, B. Powers, E. D. Berger, and A. Guha. Not so fast: Analyzing the performance of {WebAssembly} vs. native code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 107–120, 2019.

[23] A. Jangda, B. Powers, A. Guha, and E. Berger. Mind the gap: Analyzing the performance of webassembly vs. native code. *arXiv preprint arXiv:1901.09056*, 2019.

[24] D. Lehmann. *Program Analysis of WebAssembly Binaries*. PhD thesis, Universität Stuttgart, Stuttgart, Germany, July 2022. Doctoral dissertation.

[25] D. Lehmann and M. Pradel. Wasabi: A framework for dynamically analyzing webassembly. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 1045–1058, New York, NY, USA, 2019. Association for Computing Machinery.

[26] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, page 65–74, New York, NY, USA, 2007. Association for Computing Machinery.

[27] R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush. Engineering record and replay for deployability. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 377–389, Santa Clara, CA, July 2017. USENIX Association.

[28] R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush. Engineering record and replay for deployability. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 377–389, Santa Clara, CA, July 2017. USENIX Association.

[29] S. Padmanabhan and P. Jha. Webassembly at ebay: A real-world use case. `https://tech.ebayinc.com/engineering/webassembly-at-ebay-a-real-world-use-case/`, 2020. Retrieved Februar 9, 2024.

[30] G. Richards, A. Gal, B. Eich, and J. Vitek. Automated construction of javascript benchmarks. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, page 677–694, New York, NY, USA, 2011. Association for Computing Machinery.

[31] A. Romano and W. Wang. Wasmview: Visual testing for webassembly applications. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 13–16, 2020.

[32] Y. Saito. Jockey: a user-space library for record-replay debugging. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 69–76, 2005.

[33] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 488–498, New York, NY, USA, 2013. Association for Computing Machinery.

[34] S. M. Srinivasan, S. Kandula, C. R. Andrews, Y. Zhou, et al. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conference, General Track*, pages 29–44. Boston, MA, USA, 2004.

[35] B. L. Titzer. A fast in-place interpreter for webassembly. *Proc. ACM Program. Lang.*, 6(OOPSLA2), oct 2022.

[36] W. Wang. Empowering web applications with webassembly: Are we there yet? In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1301–1305, 2021.

[37] Y. Yan, T. Tu, L. Zhao, Y. Zhou, and W. Wang. Understanding the performance of webassembly applications. In *Proceedings of the 21st ACM Internet Measurement Conference*, IMC '21, page 533–549, New York, NY, USA, 2021. Association for Computing Machinery.