Wasm-R^3: Creating an Executable Benchmark of WebAssembly Binaries via Record-Reduce-Replay

Motivation

Benchmarking is important for any language, and so it is for WebAssembly. In particular, benchmarks of executable programs have proven to be extremely useful for other languages, e.g., the SPEC CPU benchmarks for C/C++ and the DaCapo benchmarks for Java. Such benchmarks are useful in various ways. One application is to measure and compare the performance of virtual machines. Another application is to perform dynamic analyses, e.g., to expose particular kinds of misbehavior, or to empirically compare dynamically and statically computed call graphs.

Currently, there is no executable benchmark of WebAssembly programs. Creating such a benchmark is challenging because WebAssembly binaries usually are not executed standalone, but as part of an application. Within an application, WebAssembly code interacts with host code, e.g., JavaScript code in a client-side web application.

As a result of the interactions with host code, the WebAssembly code may see a state of the memory that was produced by the non-WebAssembly code. A concrete example:

```
i32.const 1234
i32.const 42
i32.store  ;; store 42 at address 1234
call $some_imported_function
i32.const 1234
i32.load   ;; may load a value different from 42
```

Goal

The goal of this project is to create a benchmark of executable, standalone WebAssembly binaries. Manually creating such binaries does not scale well and is hard to repeat. Instead, we envision an automatic approach based on record-and-replay.

The planned approach will consist of three phases:

1) Record:

Given a *target binary*, the record phase instruments the given binary and executes it as part of a larger application. For example, the original execution could be based on visiting and interacting with a WebAssembly-powered website.

The instrumentation adds code that intercepts all load instructions and all interactions with imported and exported calls. Assuming that the code in the binary is deterministic, this information is sufficient to replay the exact same computation.

2) Reduce:
   Naively replaying the original execution by injecting all the recorded values to load and all the interactions with code outside of the target binary imposes a huge overhead. This overhead would hinder dynamic analysis and disturb any performance measurements performed based on the benchmark.

   To avoid most of this overhead, the reduction phase of the approach will simplify the recorded trace in a way that still allows for replaying the original execution, but with fewer instructions added to the original binary. The basic idea is to first identify loads of values that are impacted by an interaction with the host code, i.e., values that when loaded during the execution differ from those one obtains when not performing calls to imported functions. Then, the approach will identify for each of these loads an earlier call of an imported function where the corresponding store could have been performed. Finally, the approach bundles multiple such stores and performs them when the imported function gets called by the target binary.

3) Replay:
   To replay the execution of the target binary, the final phase of the approach creates a new binary, called the *replay binary*, to be run together with the original target binary. The replay binary provides all functions imported by the target binary, so that the target binary can run without any other host code. Whenever the target binary calls one of these imported functions, the replay binary performs those memory stores identified in the reduction phase. As a result, the memory state observed by the target binary looks exactly like the state observed in the original execution, but without any interactions with the host code.

   The original target binary and the replay binary together yield a standalone program that can be executed without any interactions with the host, except that the host invokes the start function of the replay binary.

Before implementing the approach, it may be wise to create a benchmark of small, hand-written WebAssembly binaries and to manually perform the planned phases of the approach.

Eventually, the evaluation of the approach should apply the approach to tens (or if time allows hundreds) of WebAssembly binaries. We plan to evaluate the approach in terms of (i) replay

accuracy (i.e., whether it preserves the original behavior) and (ii) overhead (i.e., how much additional computation the replayed execution performs compared to the original execution).

Open Questions and Challenges

- Are there any unexpected sources of non-determinism that a record-and-replay technique must be aware of?
- How exactly to reduce the original trace in a way that preserves all information needed for accurate replay, while significantly reducing the overhead imposed by replaying.
- What's the best way of implementing the instrumentation parts of the idea? Options include Wasabi (with missing support for more recent WebAssembly features) and a custom binary instrumentation.
- Which, if any, of the more recent WebAssembly features to support (e.g., threads, garbage collection and its newly added types)?
- Can we accurately measure the overhead added (time spent executing, or additional cache misses) by the replay binary?

Evaluation

- Faithfulness
- Overhead/performance of recording
- Ablation study: what if we didn't reduce? Performance, trace size?

Related Work

- Automated construction of JavaScript benchmarks, OOPSLA 2011, Richards et al.
    - https://dl.acm.org/doi/10.1145/2048066.2048119
    - PDF: https://plg.uwaterloo.ca/~dynjs/jsbench-oopsla-2011.pdf
    - Also a record&replay approach to construct "smaller" benchmarks from larger applications