Wasm-R3 Concept

Jakob Getz
University of Stuttgart / KAIST

Wasm-R3

Wasm-R3 will be a set of tools for creating WebAssembly benchmarks

Why

Useful for performance mesurement, evaluation of static analyses, ...

How

Benchmarks get automatically generated from real world applications. Because then they better represents a real world program

How does it work?

Input

- Real World Website that uses WebAssembly
- User interactions with that Website

Output

Benchmark

Example

- Input: Website: `murloc.io`
- Output: Benchmark `murloc`

Benchmark

A benchmark is a folder with the replay file and all the original wasm files from the website:

```
+-- murloc
|   +-- replay.js
|   +-- wasm/
|   |   +-- bin1.wasm
|   |   +-- bin2.wasm
```

It is also an executable program that you can run like this:

```
$ node murloc/replay.js
```

Benchmark creation

1. Instrument Website (wasm)
2. Interact with instrumented website (Recording)
3. Collect trace information (Recording)
4. Generate replay binary from trace
5. Package

1. Instrument Website

- Opening a website without Wasm-R3

```
website = { html, js, wasm }
server.send(website)
browser.receive(website)
```

- Opening a website with Wasm-R3

```
website = { html, js, wasm }
server.send(website)
instrumenter.receive(website)
instrumenter.instrument(&website.wasm) // this only affects .wasm files
instrumenter.send(website)
browser.receive(website)
```

2/3. Interact with Website and collet trace (Recording)

When a user interacts with a website certain events happen in the code of the website. For example a specific function `a` gets called or a function `b` writes a value `c` to memory.
We are interested in specific events that happen inside of the wasm code, e.g. when a wasm module calls an imported function.
Because the wasm of the website is instrumented now we can preserve information about those events in traces.

Example

- Event: wasm function `a` calls imported function `b`
- Trace of events:

```
EVT call, timestamp, caller a, callee b, mem [ ... ], tables [ ... ]
EVT return, timestamp, return_value, mem [ ... ], tables [ ... ]
```

```
type Event = {
    type: "call" | "return" | "called" | "exit_func",
    timestamp: Timestamp, // Later I will ignore this for simplicity
    memState: byte[]?,
    tableState:  Funcref[][]?,
    typeSpecificInfo: Object?
}
```

4. Generate replay binary

A replay binary is an application that takes the role of the user and not-wasm code and interacts with the wasm in exactly the same way as the user and other non-wasm components did during recording.
From the trace we collected we create a file `` `replay.js` ``

- Replay binary should be as minimal as possible
  - as small as possible (optimise / reduce)
  - as fast as possible (maybe parts of the replay binary could be also in wasm?)

5. Package

Package wasm files and replay binary into a benchmark folder

🎉 DONE

How to generate replay binary

The replay binary generator needs the trace and the original wasm instantiation code as input. Because it needs to instanciate the wasm in the same way as the original application did.
How to generate the trace? What wasm constructs to consider?

2 constructs to consider (Wasm 2.0, complete?):

1. Call to an imported function
2. Call by hostcode of an exported function

1. Call to an impored function

```
(module
    (import $b "js" "b"
        (param i32) (result i32)
    )
    (func $a (result i32)
        i32.const 0
        call $b ;; returns 1
    )
)
```

Trace:

```
EVT call, caller $a, callee $b, params
EVT return, return 0, memory nil
```

Geneated js:

```
function b(x) {
    switch (x) {
        case 0:
            return 1
            break
    }
}
```

2. Call by hostcode of an exported function

```
(module
    (func $a
        (export "a")
        (param i32)
        local.get 0 ;; is 1
    )
)
```

Trace:

```
EVT called, $a, params 1, mem nil
EVT exit_func, $a, mem nil
```

Generated js:

```
instance.exports.a(1)
```

3. Call to an imported function (if exported memory present)

```
(module
    (import $b "js" "b")
    (func $a
        i32.const 1234
        i32.const 42
        i32.store
        call $b
        i32.const 1234
        i32.load ;; returns 69
    )
    (memory (export "mem") 1))
)
```

Trace:

```
EVT call, caller $a, callee $b, params
EVT return, return nil, mem [ ... ]
```

Generated js:

```
function b() {
    // to generate this we need some lo
    // mem modification only to the par
    instance.exports.mem[1234] = 69
}
```

Also consider different behavior for different parameters (See 1. Call to an imported function)

4. Call by hostcode of an exported function (if exported memory is present)

```
(module
    (func $a (export "a")
        i32.const 1234
        i32.load ;; returns 42
    )
    (memory (export "mem") 1)
)
```

Trace:

```
EVT called, $a, params nil, mem [ ... ]
EVT exit_func, $a, mem [ ... ]
```

Generated js:

```
instance.exports.mem[1234] = 42
instance.exports.a()
```

5. Call to an imported function (if exported table is present)

```
(module
    (import $d "js" "d")
    (func $a
        i32.const 0
        call_indirect ;; calls $b
        call $d
        i32.const 0
        call_indirect ;; calls $c
    )
    (func $b (export "b"))
    (func $c (export "c"))
    (table $table (export "table") 1))
    (elem $table (i32.const 0) $b)
)
```

Trace:

```
EVT call, caller $a, callee $d, params
EVT return, return nil, tables 0 $c
```

Generated js:

```
function d() {
    instance.exports.table.modify.set(0
}
```

Also consider different behavior for different parameters (See 1. Call to an imported function)

6. Call by hostcode of an exported function (if exported table is present)

```
(module
    (func $a (export "a")
        i32.const 0
        call_indirect ;; calls $c
    )
    (func $b (export "b"))
    (func $c (export "c"))
    (table $table (export "table") 1))
    (elem $table (i32.const 0) $b)
)
```

Trace:

```
EVT called, $a, params nil, tables 0 $c
EVT exit_func, $a, tables 0 $c
```

Generated js:

```
instance.exports.table.set(0, instance.
instance.exports.a()
```

Question to myself: What happens if host code puts function in table that is defined in the host code itself??

Another case to consider

Nondeterministic host functions

```
(module
    (import $b "js" "b")
    (func $a
        i32.const 1234
        i32.const 42
        i32.store
        call $b
        i32.const 1234
        i32.load ;; returns 69
        call $b
        i32.const 1234
        i32.load ;; returns 420
    )
    (memory (export "mem") 1))
)
```

Generated js:

```
let callCounter_b = 0
function b() {
    switch (callCounter_b) {
        case 0:
            instance.exports.mem[1234]
            break
        case 1:
            instance.exports.mem[1234]
            break
    }
    callCounter_b++;
}
```

Actually...

We generate way to much trace info and replay code
so we wanna **reduce**

Example

```
(module
    (import $c "js" "c")
    (func $a (export "a")
        i32.const 1234
        i32.const 42
        i32.store
        call $c
        i32.const 1234
        i32.load ;; returns 69
    )
    (func $b (export "b")
        i32.const 1234
        i32.const 69
        i32.store
    )
    (memory (export "mem") 1))
)
```

Host code:

```
function c() {
    instance.exports.b()
}
instance.exports.a()
```

Trace:

```
EVT called, $a, params nil, mem [ ... ]
EVT call, caller $a, callee $c, params
EVT called, $b, params nil, mem [ ... ]
EVT exit_func, $b, mem [ ... ]
EVT return, return nil, mem [ ... ]
```

Example cont.

Trace:

```
EVT called, $a, params nil, mem [ ... ]
EVT call, caller $a, callee $c, params nil, mem [ ... ]
EVT called, $b, params nil, mem [ ... ]
EVT exit_func, $b, mem [ ... ]
EVT return, return nil, mem [ ... ]
```

Generated js:

```
function c() {
    instance.exports.mem[1234] = 42 // redundant
    instance.exports.b()
    instance.exports.mem[1234] = 69 // redundant
}
```

There is more potential

I did not deeply investigate where there is room for further optimisation

How to instrument the wasm

Wasabi seems to provide all features I need.

- `` `EVT called` `` in wasabi: `` `begin` `` (if type == "function")
- `` `EVT exit_func` `` in wasabi: `` `return_` ``
- `` `EVT call` `` in wasabi: `` `call_pre` ``
- `` `EVT return` `` in wasabi: `` `call_post` ``

Inside of the hooks we can access exported table and memory like this.

```
Wasabi.module.exports.table
Wasabi.module.exports.memory
```

2.0 features like `` `Multiple tables` `` and `` `Multiple values` `` seem to be supported

That’s All