COP701: Assignment 3
WebAssembly

1 Introduction

WebAssembly [1] (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable target for compilation of high-level languages like C/C++/Rust, enabling deployment on the web for client and server applications. WebAssembly 1.0 has shipped in 4 major browser engines: Firefox, Google Chrome, Safari and Microsoft Edge.

WebAssembly offers compact representation, efficient validation and compilation, and safe low to no-overhead execution. WebAssembly is an abstraction over modern hardware, making it language-, hardware-, and platform-independent, with use cases beyond just the web.

2 Problem Statement

- Analyzing the Performance of WebAssembly vs. Native Code [2].
- Analyzing the Performance of WebAssembly on modern browsers [3].
- Generating instruction traces from a WebAssembly program.

3 Experimental Setup

Benchmarks: Polybench suite [4]. It consists of 30 numerical programs. The source code can be downloaded from the cited link. Download the PolyBench/C 4.1 version.
Emscriptem: This program is used to compile a C/C++ program to WebAssembly. `https://emscripten.org/docs/getting_started/downloads.html`
Instrumentation: Wasabi [5] is a tool for dynamically analyzing WebAssembly. It is available at `https://github.com/danleh/wasabi`.

4 Deliverables - Part 1 (20th October)

4.1 WebAssembly vs. Native Code

- **Experiment 1 - Relative Execution Time:** Run the PolyBench suite on the native hardware (your system) and note down the runtime. Compile the PolyBench suite to WebAssembly using *Emscriptem.* Run it on a web-browser and note down the runtime. Plot a bar graph showing the relative execution times for all the benchmarks (similar to [2]).
- **Experment 2 - Instruction Mix Analysis:** Use the *Pin tool* [6] to instrument the native code and obtain the breakup of different types of instructions. Similarly, use the *Wasabi tool* [5] to instrument the WebAssembly code to obtain the instruction breakup. Types: *load, store, arithmetic/logical-int, arithmetic/logical-float, branch, register transfer,* and *nop.* Plot a bar graph.
- **Experiment 3 - Instruction Count:** Plot a bar graph showing the relative number of dynamic instructions for WebAssembly vis-a-vis Native Code.
- **Experiment 4 - Hot Code:** Identify the functions in the code that are run frequently and plot their dynamic instruction count.
- Repeat the experiments for different compiler optimizations O0, O1, O2 and O3. Report the details of the native hardware: DRAM size, CPU frequency, and the number of cores. Report the versions of the Operating System, C compilers, and the web-browser.

4.2 WebAssembly on modern browsers

Repeat all the experiment in Section 4.1 on at least two web-browsers from the following: *Firefox, Google Chrome, Safari and Microsoft Edge*

4.3 Report

Comment on the graphs obtained.

5 Deliverables - Part 2 (10th November)

The goal of this part is to generate dynamic instruction traces of a WebAssembly execution. Each line in the trace file should be of the following format.

$< pc > \ < type > \ < value >$

where,

- **pc:** program counter of the instruction (decimal format)

- **type:** type of the value
- **value:** dissemble of the instruction/ address

Type	Value	Remarks
27	assembly instruction	refer to the examples
2	load address	address of the load instruction in decimal format
3	store address	address of the load instruction in decimal format
4	branch target pc	when branch is taken
5	branch target pc	when branch is not taken

5.1 Examples

1. **Simple instructions:**
   $< pc >$ 27 $< instruction >$: 2147500552 27 add a0, a0, a4
2. **Load instructions:** There will be two lines in the trace corresponding to a load instruction. First line for the instruction and the second line for the load address.
   $< pc >$ 27 $< instruction >$: 2147499372 27 lw a5, 0(s8)
   $< pc >$ 2 $< address >$:2147499372 2 4184
3. **Store instructions:** There will be two lines in the trace corresponding to a store instruction. First line for the instruction and the second line for the store address.
   $< pc >$ 27 $< instruction >$: 2147499442 27 sd a0, 32(sp)
   $< pc >$ 3 $< address >$:2147499442 3 2147540416
4. **Branch taken instructions:** There will be two lines in the trace corresponding to a branch instruction. First line for the instruction and the second line for the branch target value.
   $< pc >$ 27 $< instruction >$: 2147495402 27 j pc + 0x4ae8
   $< pc >$ 4 $< branch\ target\ pc >$:2147495402 4 2147514578
5. **Branch not-taken instructions:** There will be two lines in the trace corresponding to a branch instruction. First line for the instruction and the second line for the branch target value.
   $< pc >$ 27 $< instruction >$: 2147499444 27 beqz a5, pc + 12
   $< pc >$ 5 $< branch\ target\ pc >$:2147499444 5 2147499456

5.2 Input to your program

./run.sh $< trace\ size >$ $< program\ name >$ $< program\ arguments >$

- Argument 1: Number of dynamic instructions in the trace (trace size). Example: 1000, in this case your program should exit after generating a trace for 1000 dynamic instructions. If the value of this argument is -1 then generate a trace for the whole program.

- Argument 2: The path of the program for which the trace has to be generated, followed by the arguments of this program.

5.3 Output format

Since the size of the trace would be large, your program should directly generate a compressed trace in a *tar.gz* format. The name of the output file should be *program_name.tar.gz*.

5.4 Tools

You can use Wasabi [5] or Pywasm [7] to generate the instruction traces.

5.5 WebAssembly to x86 translator

We need the traces only in the x86 assembly format. You need to translate each instruction in the WebAssembly into a x86 assembly format. For example: *i64.add* in WebAssembly translates to *add* in x86.

References

[1] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *ACM SIGPLAN Notices*, volume 52, pages 185–200. ACM, 2017.

[2] Abhinav Jangda, Bobby Powers, Emery D Berger, and Arjun 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.

[3] David Herrera, Hangfen Chen, Erick Lavoie, and Laurie Hendren. Webassembly and javascript challenge: Numerical program performance using modern browser technologies and devices. Technical report, Technical Report. Technical report SABLE-TR-2018-2. Montréal, Québec, Canada, 2018.

[4] Polybench benchmark suite. `http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/`, 2019.

[5] Daniel Lehmann and Michael 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*, pages 1045–1058. ACM, 2019.

[6] Pin tool. `https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`, 2019.

[7] pywasm. `https://pypi.org/project/pywasm/`, 2019.