Quantum Annealers as Continuous Testing Automation Backends for Classical Web Code

Marcus Edwards[1], Dr. Atefeh Mashatan[2], Dr. Shohini Ghose[3]

Physics and Astronomy, University of Waterloo, Waterloo, ON[1],Cybersecurity Research Lab, Ted Rogers School of Information Technology Management, Ryerson University[2] , Physics and Computer Science, Wilfrid Laurier University, Waterloo, ON[3]

Introduction

We present a methodology and software package for verifying WebAssembly (WASM) web code using quantum annealers. Executing a simulation of a WASM function on a quantum annealer enables fast edge-case detection which can be used to verify the correctness of the function. The method benefits from a speedup due to quantum tunneling in the annealer. This suggests an efficiency improvement over the analogous classical sampling techniques being adopted by web software companies.

Background

Efforts have been made to generalize the classical logic systems that can be optimized on D-Wave's annealers. Most programming tools deal with very low-level logic systems. For example, macro assemblers called QMASM and qbsolv have syntax similar to assembly languages and essentially just directly specify the linear and quadratic coefficients of problems intended for D-Wave. ThreeQ.jl enables the construction of QUBOs within the Julia programming language. In April 2019, Scott Pakin tackled a formidable task and introduced a method of compiling arbitrary Verilog code through a number of steps to QMASM that can be executed on a D-Wave system. This is the state-of-the-art in quantum annealer based simulation of classical programs written in traditional programming languages.

D-Wave also provides their own Python library for programming their annealers. This library has recently introduced some higher level methods that provide developers to ability to easily compose Hamiltonians that correspond to simulations of slightly more complex digital constructs. For example, library methods exist for creating simulations of combinational half and full adders.

Goals

We argue that a specific domain of continuous test automation tasks that is immediately relevant to the verification of the correctness and security of large-scale commercial software development projects could benefit from improvements by delegating a particular class of tests to execution on a quantum annealer using our new method.

We introduce a transpilation technology “QuantEmu” that enables the emulation of algorithms written in practical programming languages including Rust, C/C++, PHP, Python, Ruby, TypeScript and JavaScript on D-Wave's quantum annealer systems to this end. A logical next step for future work would be to provide a language agnostic test automation framework for the validation of web-based software.

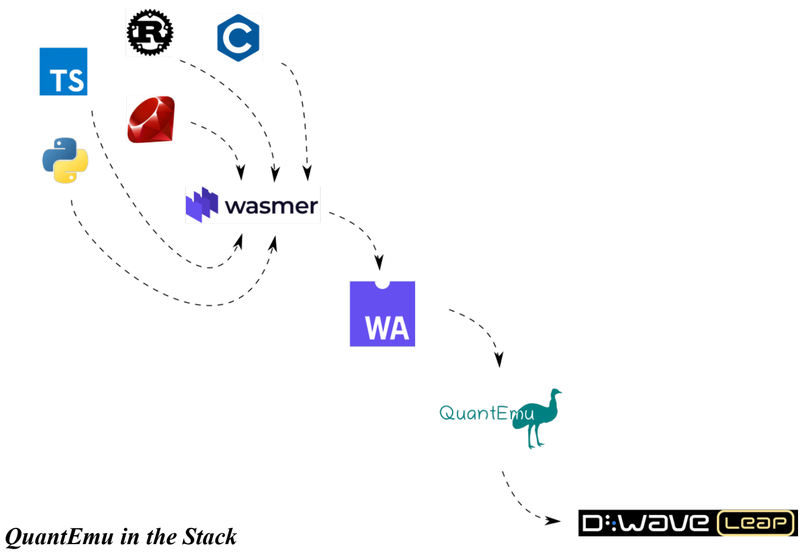

Figure 1: QuantEmu in the Stack

Methods

While WASM is more hardware agnostic than other assembly languages like x86 and we don't have to deal with assumptions in the language about execution hardware, compiling arbitrary WASM code in its entirety to QMASM is still not a reasonable endeavor. Instead, we classify types of code blocks that can be simulated either in one shot, or as a part of a multi-step validation process. A fundamental difficulty in translating WASM to QMASM is the sequential nature of WASM code. We are limited to 2048 qubits for code and data, so this defines the boundary between what WASM modules will be simulatable and what modules need to be broken up further before being simulated. Hence, our WASM transpilation process involves several intermediate steps.

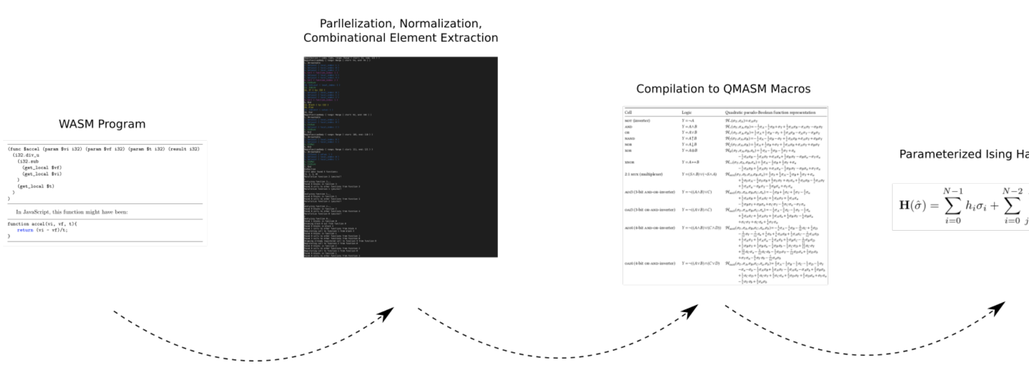

Figure 2: Transpilation Steps

Methods Continued

Dependency Collapsing for Compatibility with Annealing:

- Collapses all data, name and control dependencies into a single feed-forward data tree
- Each data dependency is mapped to a qubit
- The effectively achieves the simulation of the sequential program by trading the time dimension for additional spatial dimensions

Combinational Element Abstraction:

- Each block of WASM instructions that can be combinationally executed is extracted
- Control structures are normalized to maximize the size and number of these combinational blocks
- These blocks can be compiled individually to QMASM scripts
- Each of these is a candidate for execution on a quantum annealer

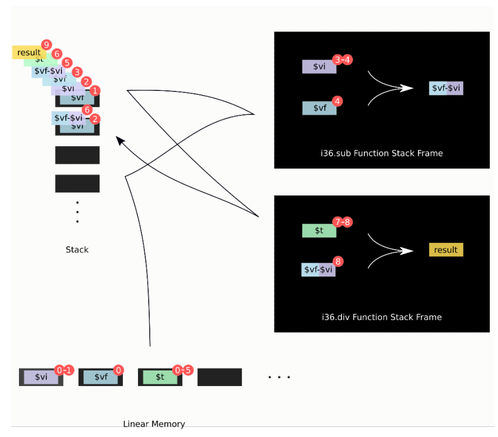

Figure 3: Dependency Tracing

Results

A demo WASM module “math.wasm” provides functions that compute the dot products of 2 and 3 dimensional vectors. This module was written for the purposes of this demonstration in the S-expression format. Passing this method to our Rust transpiler is done using the following command.

```
$ cargo run --example parallelize  ./tests/parallelization/math.wasm
```

Figure 4: Transpilation Command

The transpiler then reads the WASM file, categorizing each instruction into one of the categories: function calls, control flow instructions, data flow instructions, data creation instructions, data mutation instructions. All paths through the program are traced out. The user is asked whether to parallelize each function.

```
BeginWasm { version: 1 }
BeginSection { code: Type, range: Range { start: 10, end: 29 } }
TypeSectionEntry(FuncType { form: Func, params: [I32, I32, I32, I32, I32, I32], returns: [I32] })
TypeSectionEntry(FuncType { form: Func, params: [I32, I32, I32, I32], returns: [I32] })
EndSection
BeginSection { code: Function, range: Range { start: 31, end: 34 } }
EndSection
BeginSection { code: Export, range: Range { start: 36, end: 59 } }
ExportSectionEntry { field: "dot_three", kind: Function, index: 0 }
ExportSectionEntry { field: "dot_two", kind: Function, index: 1 }
EndSection
BeginSection { code: Code, range: Range { start: 61, end: 95 } }
BeginFunctionBody { range: Range { start: 63, end: 81 } }
1. Unreachable
2. GetLocal { local_index: 0 }
3. GetLocal { local_index: 3 }
4. I32Mul
5. GetLocal { local_index: 1 }
6. GetLocal { local_index: 2 }
7. GetLocal { local_index: 4 }
8. GetLocal { local_index: 5 }
9. Call { function_index: 1 }
10. I32Add
11. End
BeginFunctionBody { range: Range { start: 82, end: 95 } }
1. Unreachable
2. GetLocal { local_index: 0 }
3. GetLocal { local_index: 2 }
4. I32Mul
5. GetLocal { local_index: 1 }
6. GetLocal { local_index: 3 }
7. I32Mul
8. I32Add
9. End
EndSection
First pass found 2 functions:
[0, 1]
```

Figure 5: Instruction Categorization

The code is normalized using a parallelizing compilation methods inspired by Fortran’s Parallelizing Fortran Compiler (PFC), and then each combinational instruction is translated to a Boolean function that captures its behavior. Finally, the user can provide constraints and the resulting data structure is handed off for conversion to a QUBO and Hamiltonian parameters.

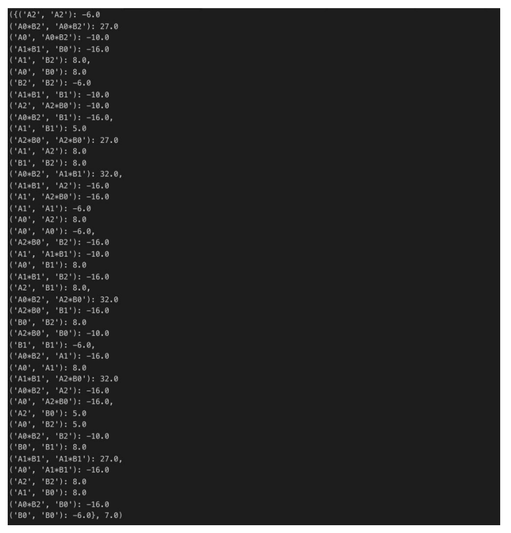

Figure 6: Ising Hamiltonian Parameters

Ryerson University

TED ROGERS SCHOOL Information Technol Management

WILFRID LAURIER UNIVERSITY LAURIER Inspiring Lives.

IQC Institute for Quantum Computing

NSERC CRSNG

CYBERSECURITY RESEARCH LAB www.ryerson.ca/crl