Martin Walls

C to WebAssembly Compiler

Computer Science Tripos: Part II

Churchill College

20th February, 2023

Declaration of Originality

I, Martin Walls of Churchill College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. I am content for my dissertation to be made available to the students and staff of the University.

Signed: MWalls

Date: 20th February, 2023

Proforma

Candidate Number:	**TODO**
Project Title:	C to WebAssembly Compiler
Examination:	Computer Science Tripos: Part II
Year:	2023
Dissertation Word Count:	**TODO**[1]
Code Line Count:	**TODO**[2]
Project Originator:	Timothy M. Jones
Supervisor:	Timothy M. Jones

Original Aims of the Project

At most 100 words describing the original aims of the project.

Work Completed

At most 100 words summarising the work completed.

Special Difficulties

None.

[1] Word count computed by `texcount -inc -total -sum`

[2] Code line count computed by `cloc . -by-file -force-lang=Rust,lalrpop -list-file=.cloc`

Contents

1

Introduction

Word budget: ~500–600 words

Explain the main motivation for the project
Show how the work fits into the broad area of surrounding computer science
brief survey of previous related work
-> original emscripten (compiler LLVM to JS)
-> various compilers to wasm, including from LLVM which would do C

1.1 Background and Motivation

1.2 Project Objectives

1.3 Survey of Related Work

2

Preparation

Word budget: ~2500-3000 words

Describe the work undertaken before code was written.
-> Wasm research – include the stuff from the research doc I wrote.
-> include Relooper research here too
”Requirements Analysis” section
-> refer to appropriate software engineering techniques used in the diss
Cite new programming language learnt
Declare starting point
Explain background material required beyond IB
Researching LALRPOP - show good professional use of tools
Talk about revision control strategy, licensing of any libraries I used

2.1 Project Strategy

2.1.1 Requirements Analysis

2.1.2 Software Engineering Methodology

2.1.3 Testing

2.2 Starting Point

2.2.1 Knowledge and experience

- IB Compilers Course
- Experience with JavaScript + Python (cos I used those for runtime/testing)
- Experience writing C, my source language

2.2.2 Tools Used

- Say here that I learned Rust for this project
- Also used JavaScript for runtime + Python for testing

3

Implementation

Word budget: ~4500–5400 words

3.1 Repository Overview

I developed my project in a GitHub repository[1], ensuring to regularly push to the cloud for backup purposes. This repository is a monorepo containing both my research and documentation along with my source code.

```
    ├── headers/ ........................ Header files for the standard library functions
    │   │                                 I implemented
    │   ├── stdio.h
    │   └── ...
    ├── runtime/ ........................ NodeJS runtime environment
    │   ├── stdlib/ ..................... Implementations of standard library functions
    │   │                                 in JS
    │   ├── run.mjs
    │   └── ...
    ├── src/ ............................ The source code for the compiler, explained be-
    │   │                                 low
    │   └── ...
    ├── tests/ .......................... Test specification files
    │   └── ...
    └── tools/
        ├── profiler.py ................. Code to plot stack usage profiles
        └── testsuite.py ................ Test runner
src/
    ├── back_end/
    ├── data_structures/
    ├── front_end/
    ├── middle_end/
    ├── program_config/
    ├── relooper/
    ├── fmt_indented.rs
    ├── id.rs
    ├── lib.rs
    └── main.rs
```

[1] https://github.com/martin-walls/cam-part-ii-c-webassembly-compiler

```
└── preprocessor.rs
```

Finish this. Will have to see if it'll be better to have comments on the right of dirs, or to highlight the main structure below

3.2 System Architecture

Compiler Pipeline overview – include the diagram here

3.3 Front End: Lexer and Parser

Describe what was actually produced.
Describe any design strategies that looked ahead to the testing phase, to demonstrate professional approach

- wrote parser grammar
Talk about avoiding ambiguities - eg. dangling else - by using Open/Closed statement in grammar
Talk about my `interpret_string` implementation, to handle string escaping. Implemented using an iterator.
- wrote custom lexer - cos typedefs make C context-sensitive, so handle them as we see them so they don't get mixed with identifiers
- created AST representation
Talk about structure of my AST
Talk about how I parsed type specifiers into a standard type representation. Used a bitfield to parse arithmetic types, cos they can be declared in any order.

Describe high-level structure of codebase.
Say that I wrote it from scratch.
-> mention LALRPOP parser generator used for .lalrpop files

3.4 Middle End: Intermediate Representation

- Defined my own three-address code representation
- for every ast node, defined transformation to 3AC instructions
- created IR data structure to hold instructions + all necessary metadata
- Talk about auto-incrementing IDs - abstraction of the Id trait and generic IdGenerator struct
- handled type information - created data structure to represent possible types
- making sure instructions are type-safe, type converting where necessary - talk about unary/binary conversions, cite the C reference book
- Compile-time evaluation of expressions, eg. for array sizes
- Talk about the Context design pattern I used throughout – maybe research this and see if it's been done before?

3.5 The Relooper Algorithm

cite Emscripten [1]

3.6 Back End: Target Code Generation

3.7 Optimisations

3.7.1 Unreachable Procedure Elimination

3.7.2 Tail-Call Optimisation

Defn of tail-call optimisation
Why do the optimisation

3.8 Summary

4

Evaluation

Word budget: ~2000–2400 words

”Signs of success, evidence of thorough and systematic evaluation”
- How many of the original goals were achieved?
- Were they proved to have been achieved?
- Did the program really work?
Answer questions posed in the introduction
use appropriate techniques for evaluation, eg. confidence intervals

Talk about how I wrote my test script to automatically compare with GCC.
Talk about the test programs I used.

4.1 Success Criteria

4.2 Testing

4.3 Correctness

4.4 Performance Impacts of Optimisations

4.4.1 Profiling

Talk about how I implemented the stack profiler

4.4.2 Tail-Call Optimisation

My implementation of tail-call optimisation was successful in reusing the existing function stack frame for tail-recursive calls. The recursion is converted into iteration within the function, eliminating the need for new stack frame allocations. Therefore, the stack memory usage remains constant rather than growing linearly with the number of recursive calls.

To evaluate the optimisation, I used the following function that uses tail-recursion to compute the sum of the first n integers.

```c
long sum(long n, long acc) {
    if (n == 0) {
        return acc;
    }
    return sum(n - 1, acc + n);
}
```

Listing 4.1: Tail-recursive function to sum the integers 1 to n

Figure 4.1 compares the stack memory usage with tail-call optimisation disabled and enabled. Without the optimisation, the stack size grows linearly with n. When running the program with $n = 500$, a stack size of 46.3 kB is reached. When the same program is compiled with tail-call optimisation enabled, only 298 bytes of stack space are used. This is a 99.36 % reduction in memory usage.

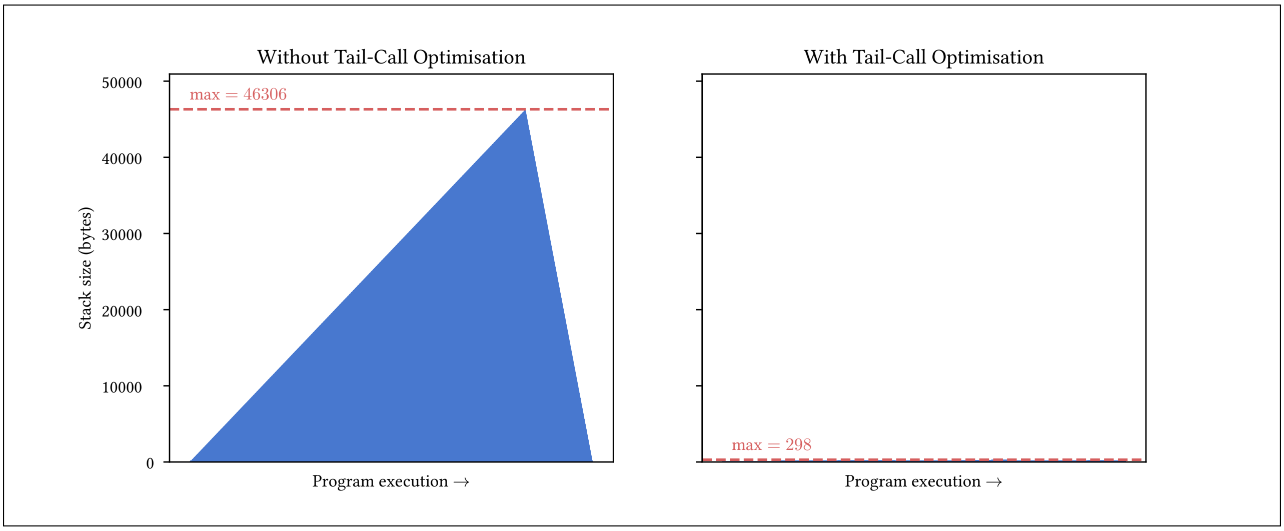


Figure 4.1: Stack usage for calling `sum(500, 0)` (see **Listing 4.1**)

4.4.3 Stack Allocation Policy

The stack allocation policy that I implemented was successful in reducing the amount of stack memory used.

From my test programs, the highest gain was xxx and the average gain was xxx. The amount of different made depended on how many temp vars there were, and how much they clashed with each other.

4.5 Summary

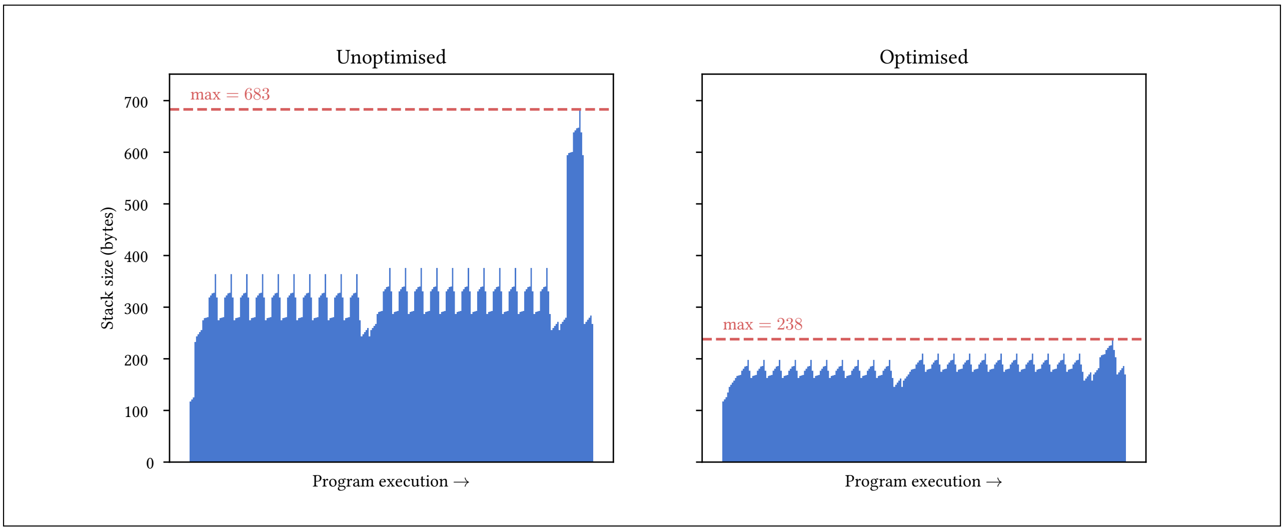


Figure 4.2: Stack profile

5

Conclusions

Word budget: ~500–600 words

Likely short, may well refer back to the introduction. Reflection on lessons learned, anything I'd have done differently if starting again with what I know now.
First paragraph should reiterate what the project was about.
Summarise how my evaluation answered the questions this project was asking
Can briefly outline any ideas for further work

5.1 Project Summary

5.2 Lessons Learned

5.3 Further Work

- implement more of stdlib, eg. malloc() and free()

Bibliography

[1] Alon Zakai. *Emscripten: An LLVM-to-JavaScript Compiler*. Mozilla, 2013. URL: `https://raw.githubusercontent.com/emscripten-core/emscripten/main/docs/paper.pdf`.

Index

Appendix A
Project Proposal

The original project proposal is included on the following pages.

Part II Project Proposal: C to WebAssembly Compiler

Martin Walls

October 2022

Overview

With the web playing an ever-increasing role in how we interact with computers, applications are often expected to run in a web browser in the same way as a traditional native application. WebAssembly is a binary code format that runs in a stack-based virtual machine, supported by all major browsers. It aims to bring near-native performance to web applications, with applications for situations where JavaScript isn't performant enough, and for running programs originally written in languages other than JavaScript in a web browser.

I plan to implement a compiler from the C language to WebAssembly. C is a good candidate for this project because it is quite a low-level language, so I can focus on compiler optimisations rather than just implementing language features to make it work. Because C has manual memory management, I won't have to implement a garbage collector or other automatic memory management features. Initially I will provide support for the stack only, and if time allows I will implement `malloc` and `free` functionality to provide heap memory management.

I will compile a subset of the C language, to allow simple C programs to be run in a web browser. A minimal set of features to support will include arithmetic, control flow, variables, and functions (including recursion). I won't initially implement linking, so the compiler will only handle single-file programs. This includes not linking the C standard library, so I will provide simple implementations of some of the standard library myself, as necessary to provide common functionality such as `printf`.

I will use a lexer and parser generator to do the initial source code transformation into an abstract syntax tree. I will focus this project on transforming the abstract syntax tree into an intermediate representation—where optimisations can be done—and then generating the target WebAssembly code.

I plan to write the compiler in Rust, which is memory safe and performant, and has lexer/parser generators I can use.

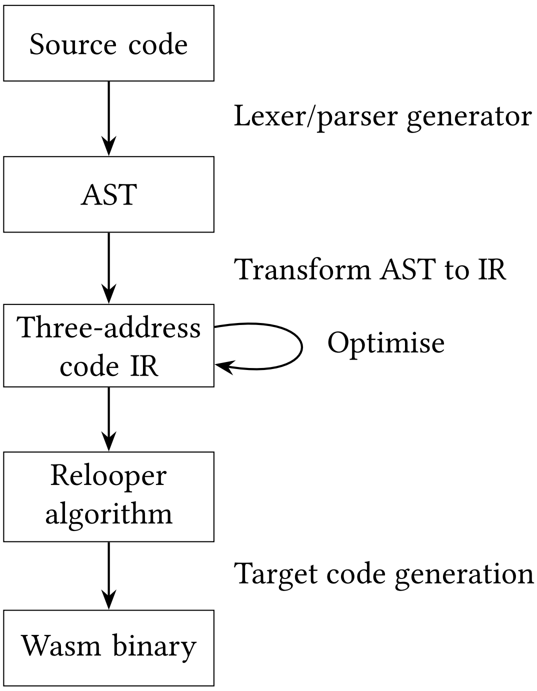
To test and evaluate the compiler, I will write small benchmark programs that individually test each of the features and optimisations I add. For example, I will use the Fibonacci program to test recursion. I will also test it with Conway's Game of Life, as an example of a larger program, to test and evaluate the functionality of the compiler as a whole.

I will use a three-address code style of intermediate representation, because this lends itself to perform optimisations more easily. For example it's easier to see the control flow in three-address code compared to a stack-based representation. To transform from abstract syntax tree to the intermediate representation, this will involve traversing the abstract syntax tree recursively, and applying a transformation depending on the type of node to three-address code.

To transform from the intermediate representation to WebAssembly, I will need to convert the three-address code representation into a stack-based format, since WebAssembly is stack-based. This stack-based format will have a direct correspondence to WebAssembly instructions, so the final step of the compiler will be writing out the list of program instructions to a WebAssembly binary file.

C allows unstructured control flow (e.g. `goto`), whereas WebAssembly only supports structured control flow. Therefore I will need a step in the compiler to transform unstructured to structured control flow. One algorithm to do this is the Relooper algorithm, which was originally implemented as part of Emscripten, a LLVM to JavaScript compiler[1].

Compiler pipeline overview



Starting point

I don't have any experience in writing compilers beyond the Part IB Compiler Construction course. I haven't previously used any lexer or parser generator libraries. I've briefly looked at Rust over the summer, but haven't written anything other than simple programs in it.

I have briefly looked up the instruction set for WebAssembly and have written a single-function program that does basic arithmetic, in WebAssembly text format. I used `wat2wasm` to convert this to a WebAssembly binary and ran the function using JavaScript.

[1] `https://github.com/emscripten-core/emscripten/blob/main/docs/paper.pdf`

I have briefly researched lexer and parser generators to see what's out there and to help decide on which language to write my compiler in, but I haven't used them before.

Success criteria

The project will be a success if:

- The program generates an abstract syntax tree from C source code.
- The program transforms the abstract syntax tree into an intermediate representation.
- The program uses the Relooper algorithm to transform unstructured to structured control flow.
- The program generates WebAssembly binary code from the intermediate representation.
- The compiler generates binary code that produces the same output as the source program.

Optimisations

First I will implement some simple optimisations, before adding some more complicated ones.

One of the simple optimisations I will implement is peephole optimisation, which is where we look at short sections of code and match them against patterns we know can be optimised, then replacing them with the optimised version. For example, redundant operations can be removed, such as writing to the same variable twice in a row (ignoring the first value written), or a stack push followed immediately by a pop. Null operations (operations that have no effect, such as adding zero) can also be removed.

Constant folding is another quite simple optimisation that performs some arithmetic at compile time already, if possible. For example, the statement `x = 3 + 4` can be replaced by `x = 7` at compile time; there is no need for the addition operation to be done at runtime.

These optimisations will be run in several passes, because doing one optimisation may then allow another optimisation to be done that wasn't previously available. The optimisation passes will run until no further changes are made.

The stack-based peephole optimisations (such as removing pushes directly followed by a pop) will be done once the three-address code representation has been transformed into the stack-based format in the final stage.

A more complicated optimisation to add will be tail-call optimisation, which removes unnecessary stack frames when a function call is the last statement of a function.

Other harder optimisations are left as extensions to the project.

Extensions

Extensions to this project will be further optimisations. These optimisations are more complicated and will involve more analysis of the code.

One optimisation would be dead-code elimination, which looks through the code for any variables that are written to but never read. Code that writes to these variables is removed, saving processing power and space.

Another optimisation would be unreachable-code elimination, where we perform analysis to find blocks of code that can never be executed, and removing them. This will involve control flow analysis to determine the possible routes the program can take.

Evaluation

To test and evaluate the compiler, I will use it to compile a variety of different programs. Some of these will be small programs I will write to specifically test the features and optimisations of the compiler individually. I will also write a larger test program to evaluate the compiler as a whole.

In addition, I will use some pre-existing benchmark programs to give a wider range of tests. For example, cBench is a set of programs for benchmarking optimisations, which I could choose appropriate programs from. The source for cBench is no longer available online, but my supervisor is able to give me a copy of them.

For each of these, I will verify that the generated WebAssembly code produces the same output as the source program when run.

To evaluate the impact of the optimisations, I will run the compiler once with optimisations enabled and once with them disabled, on the same set of programs. I will then benchmark the performance of the output program to identify the impact of the optimisations on the program's running time, and I will also compare the size of the two programs to assess the impact on storage space.

Work Plan

1	**14th - 28th Oct**	Preparatory research, set up project environment, including toolchain for running compiled WebAssembly. I will research the WebAssembly instruction set. I will also write test C programs for Fibonacci and Conway's Game of Life. To help with my WebAssembly research, I will implement the same Fibonacci program in WebAssembly by hand. ***Milestone deliverable****: I will write a short LaTeX document explaining the WebAssembly instruction set, from the research I do.* *C programs of Fibonacci and Conway's Game of Life, and a WebAssembly implementation of Fibonacci.*

2	**28th Oct - 11th Nov**	Lexer and parser generator implementation. This will involve writing the inputs to the lexer and parser generators to describe the grammar of the source code and the different types of tokens. ***Milestone deliverable****: Lexer and parser generator inputs. The compiler will be able to generate an abstract syntax tree (AST) representation from a source program.*
3	**11th - 25th Nov**	Implementation of transforming the AST into the intermediate representation. This will require defining the intermediate code to generate for each type of node in the AST. ***Milestone deliverable****: The compiler will be able to generate an intermediate representation version from a source program.*
4	**25th Nov - 9th Dec**	Researching and implementing the Relooper algorithm. ***Milestone deliverable****: The compiler will be able to transform unstructured control flow into structured control flow using the Relooper algorithm. I will also write a short LaTeX document describing the algorithm.*
5	**9th - 23rd Dec**	Implementation of target code generation from intermediate representation. For each type of instruction in the intermediate representation, I will need to define the transformation that generates WebAssembly from it. ***Milestone deliverable****: The compiler will be able to generate target code for a source program. The generated WebAssembly will be able to be run in a web browser.*
		Two weeks off over Christmas
6	**6th - 20th Jan**	*(I'll be more busy during the first week of this with some extracurricular events before term.)* Slack time to finish main implementation if necessary. Implement some peephole optimisations (how many I do here depends on how much of the slack time I need). ***Milestone deliverable****: The basic compiler pipeline will be complete. Some peephole optimisations will be implemented.*
7	**20th Jan - 3rd Feb**	Write progress report. Continue implementing optimisations, in particular implementing tail-call optimisation. ***Milestone deliverable****: Completed progress report. (Deadline 03/02)*
8	**3rd - 17th Feb**	*(I'll be more busy here with extra-curricular events.)* Slack time to finish main optimisations if necessary. If time allows, work on extension optimisations.

		***Milestone deliverable**: The compiler will be able to generate target code with optimisations applied. Evidence to show the impact of the optimisations.*
9	**17th Feb - 3rd Mar**	Evaluate the compiled WebAssembly using a variety of programs (as described above), including correctness and impact of optimisations. Write these evaluations into a draft evaluation chapter. ***Milestone deliverable**: Draft evaluation chapter.*
10	**3rd - 17th Mar**	Write introduction and preparation chapters. ***Milestone deliverable**: Introduction and preparation chapters.*
11	**17th - 31st Mar**	Write implementation chapter. ***Milestone deliverable**: Implementation chapter.*
12	**31st Mar - 14th Apr**	Write conclusions chapter and finish evaluations chapter. ***Milestone deliverable**: Evaluations and conclusions chapter. First draft of complete dissertation.*
13	**14th - 28th Apr**	Adjust dissertation based on feedback. ***Milestone deliverable**: Finished dissertation.*
14	**28th Apr - 12 May**	Slack time in two weeks up to formal deadline, to make any final changes. ***Milestone deliverable**: Final dissertation submitted. (Deadline 12/05)*

Resource declaration

I will primarily use my own laptop for development. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.

My laptop specifications are:

- Lenovo IdeaPad S540
- CPU: AMD Ryzen 7 3750H
- 8GB RAM
- 2TB SSD
- OS: Fedora 35

I will use Git for version control and will regularly push to an online Git repository on GitHub. I will clone this repository to the MCS and regularly update the clone, so that if my machine fails I can immediately continue work on the MCS.