1

Evaluation

"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.

1.1   Success Criteria

The success criteria for my project, as defined in my project proposal, are:

- 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.

All of my success criteria have been met. The first four criteria correspond to the main stages of the compiler pipeline, respectively. The correctness of this pipeline is verified by the correctness of the generated binary code. The generated binary would not be correct if any of the stages had not been successful.

I used a variety of test programs to verify the success of the fifth criteria. This is described in **Section 1.2**.

1.2 Testing

I wrote a suite of test programs in C to evaluate the correctness and performance of my compiler. There were two types of program:

- 'Unit test' programs, which test a specific construct in the C language;

- Full programs, which represent real workloads that the compiler would be used for.

The first type of test program are not strictly unit tests in the standard sense of the word. However, since they verify the functionality of the compiler on individual parts of the C language, they fulfil a very similar purpose.

Examples of 'full programs' include Conway's Game of Life [1], calculating the Fibonacci numbers (recursively), and finding occurrences of a substring in a string.

For all my tests, I used GCC[1] as the reference for correctness. I deemed a program to be correct if it produced the same output as when compiled with GCC. To facilitate this, I made liberal use of `printf`, to output the results of computations.

I wrote a test runner script to ensure that I maintained correctness as I continued to develop the compiler, fix bugs, and implement optimisations. This script read a directory of `.yaml` files, which described the path to each test program, and the arguments to run it with. It then compiled the program with both my compiler and with GCC, and compared the outputs. A test passed if the outputs were identical, and failed otherwise. The script reported which tests, if any, failed.

I also implemented some convenience features into the test script. The command-line interface takes an optional 'filter' argument, which can be used to run a subset of the tests whose name matches the filter. The script can also be used to run one of the test programs without comparing to GCC, printing to the standard output. This allows easier manual testing.

To prevent bugs from accidentally creeping in to my compiler, I set up the test suite to run automatically as a commit hook. This would prevent a commit from succeeding if any of the tests failed, allowing me to make corrections first. This ensured that the version of my project in source control was always correct and functioning.

1.3 Performance Impacts of Optimisations

1.3.1 Unreachable Procedure Elimination

In the context of this project, unreachable procedure elimination mainly has benefits in removing unused standard library functions from the compiled binary. When a standard library header is included in the program, the preprocessing stage inserts the entire code of that library[2]. If the program only uses one or two of the functions, most of them will be dead code. Unreachable procedure elimination is able to safely remove these.

---

[1]GCC version 11.3.1

[2]That is, the entirety of my skeleton implementation of that library, rather than the actual standard library code.

I only implemented enough of the standard library to allow my test programs to run, so the impact of this optimisation is limited by the number of functions imported. If I were to implement more of the standard library, this optimisation would become more important.

The standard library header with the most functions that I implemented was `ctype.h`. This header contains 13 functions, of which a program might normally use two or three. This is where I saw the biggest improvement from the optimisation. Programs that used the `ctype` library saw an average file-size reduction of 4.7 kB.

The other standard library headers I implemented only contained a few functions, so the impact of this optimisation was much more limited. However, as mentioned above, if I implemented more of the standard library, I would see much more of an improvement.

1.3.2 Profiling

Talk about how I implemented the stack profiler

1.3.3 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 1.1: Tail-recursive function to sum the integers 1 to n

Figure 1.1 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.

1.3.4 Stack Allocation Policy

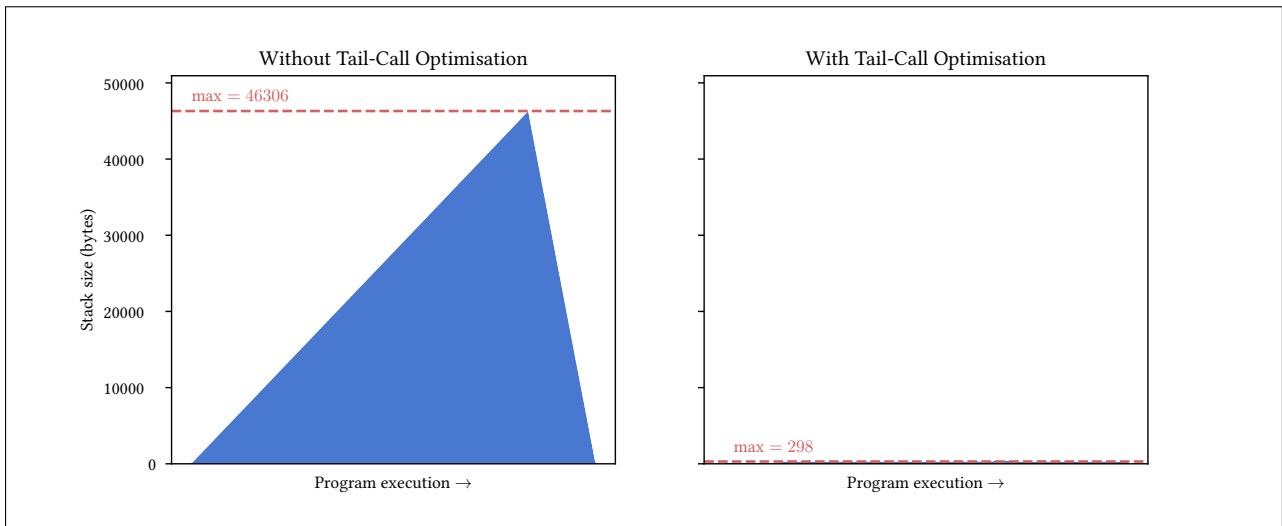The stack allocation policy that I implemented was successful in reducing the amount of stack memory used.

Figure 1.1: Stack usage for calling `sum(500, 0)` (see **Listing 1.1**)

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.
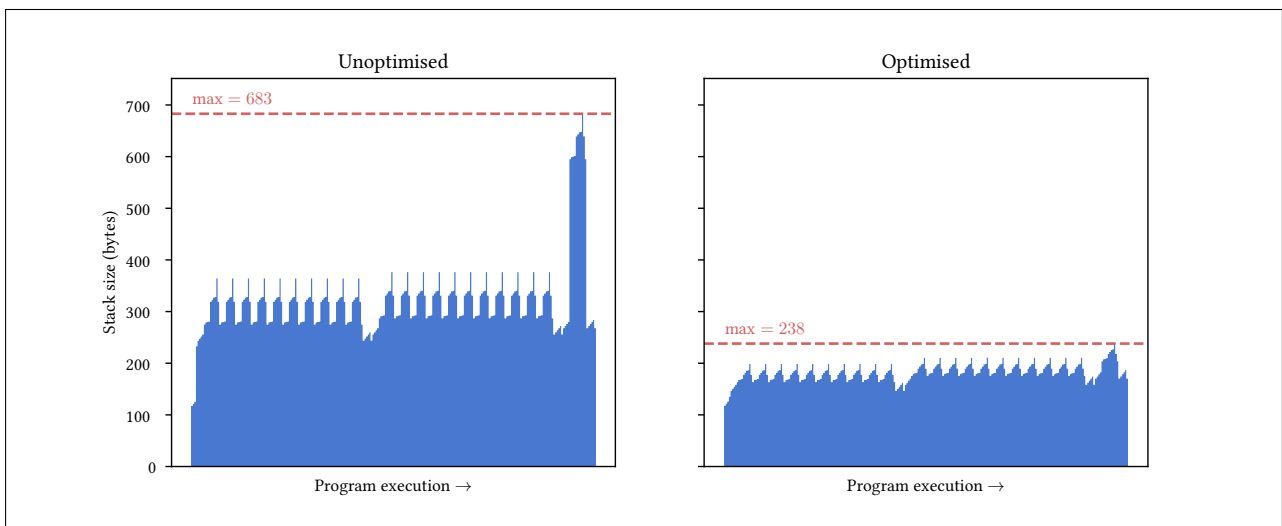

Figure 1.2: Stack profile

1.4 Summary