1 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.

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. Some of the test programs I used were sourced from Clib [2], which contains many small utility packages for C. Some of these had no external dependencies, and were useful in verifying that my compiler also worked for other people's code. Clib is licensed under the MIT license, which permits use of the software "without restriction".

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 pro-

[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.

gram only uses one or two of the functions, most of them will be dead code. Unreachable procedure elimination is able to safely remove these.

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 Stack Usage Profiling

The other two optimisations I implemented both reduced the amount of stack memory used by programs. To evaluate the effectiveness of the optimisations, I inserted profiling code when compiling the programs, to measure the size of the stack throughout program execution.

When generating instructions that move the stack pointer, the compiler additionally inserts a call to `log_stack_ptr`, which is a function imported from the JavaScript runtime. `log_stack_ptr` reads the current stack pointer value from memory and appends it to a log file.

To visualise the resulting data, I wrote a Python script to plot the stack usage log. All the figures below are generated using this profiling method.

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.

One of the functions I used to evaluate this optimisation was the function in **Listing 1.1** below, that uses tail-recursion to compute the sum of the first n integers.

```
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; a 99.36 % reduction in memory usage. Of course, the reduction depends on how many iterations of the function are run. In the non-optimised case, the stack usage is $\mathcal{O}(n)$ in the number of iterations, whereas in the optimised case it is $\mathcal{O}(1)$.

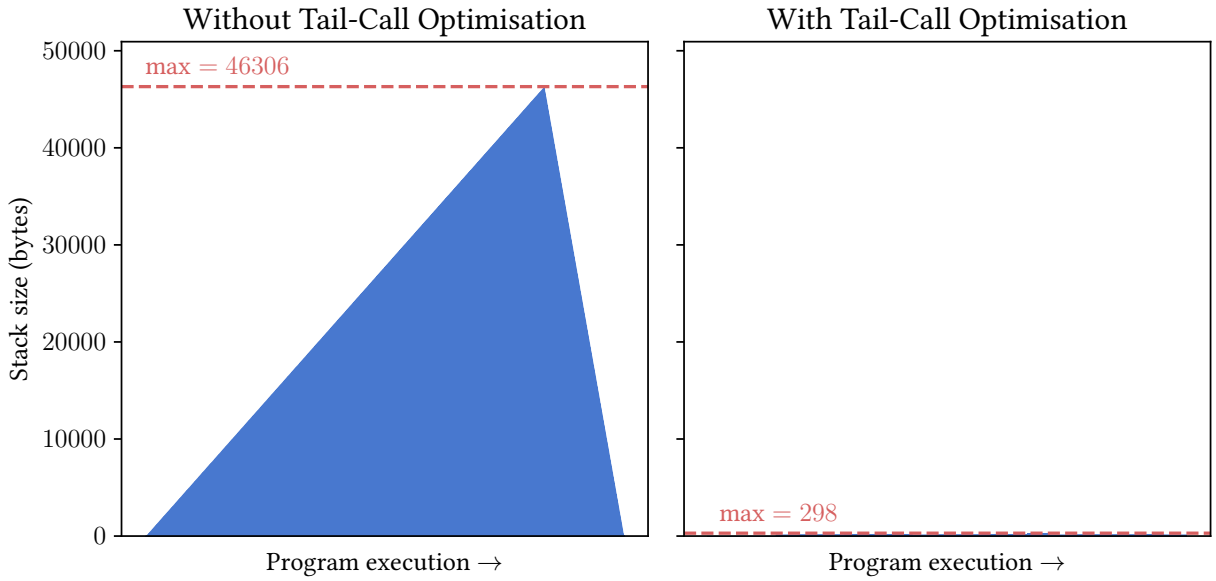

Figure 1.1: Stack usage for calling `sum(500, 0)` (see **Listing 1.1**)

When testing with large n, the non-optimised version quickly ran out of memory space, and threw an exception. In contrast, the optimised version has no memory constraint on how many iterations can be run. It successfully runs 1 000 000 levels of recursion without needing any more memory; even GCC fails to run that many.

1.3.4 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 69.73 %. The average gain was 50.28 %, with a 95 % confidence interval of $[41.37\%, 59.19\%]$.

Figure 1.2 shows the impact that this optimisation had on the different test programs. The full light-blue bars represent the stack usage of the unoptimised program, which we measure the optimised program against. The darker blue bars show the stack usage of the optimised program as a percentage of the original stack usage. Shorter bars represent a greater improvement (less memory is being used).

One of the main factors influencing the amount of improvement is the number of temporary variables generated. The more temporary variables generated, the larger each stack frame will be in the unoptimised version, and the more scope there is for the compiler to find non-clashing variables to overlap. Because temporary variables are generated locally for each instruction, the majority of

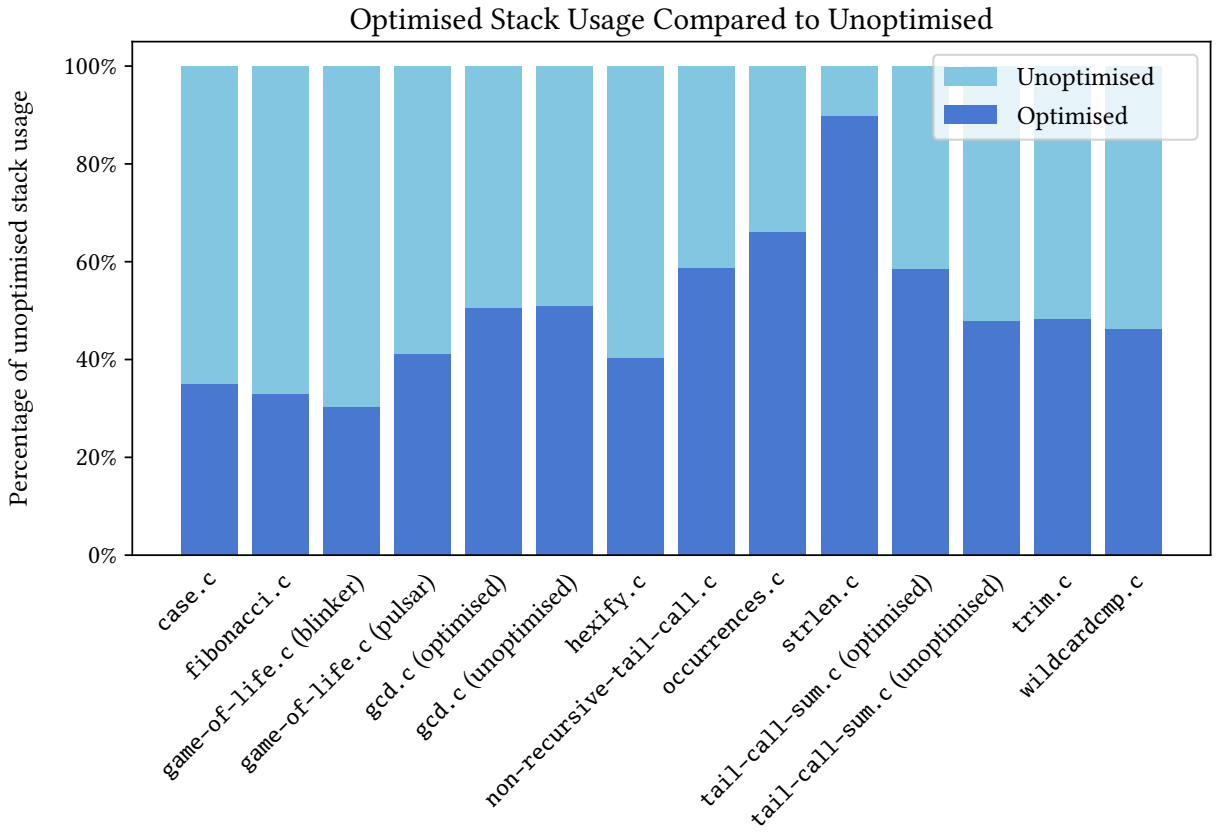

Figure 1.2: Comparing optimised stack usage to unoptimised stack usage. Shorter bars represent more improvement.

them only have short-range dependencies. Only the variables that correspond to user variables have longer-range dependencies. Therefore the temporary variables offer the compiler more options of independent variables.

The result of this is that as a function increases in its number of operations, the number of temporary variables increases, and so does the scope for optimisation that the compiler is able to exploit.

We can see the effect of this directly when we compare the stack usage of the unoptimised and optimised versions of the same program. **Figure 1.3** shows the size of the stack over the execution of a test program that converts strings to upper, lower, or camel case. Since the main stack allocations and deallocations occur on function calls and returns respectively, we can trace the execution of the program through the stack usage plot.

The program in turn calls `case_upper()`, `case_lower()`, and `case_camel()`, which corresponds to the three distinct sections of the plot.

`case_upper()` and `case_lower()` each make repeated calls to `toupper()` and `tolower()`, which corresponds to the many short spikes on the plot (one for each character in the string). Other than a `for` loop, they do not contain many operations, and therefore not many temporary variables are generated.

In contrast, `case_camel()` performs many more operations iteratively in the body of the function. **Listing 1.3** shows an extract of its body code. Even in this short section, more temporary variables are created than in the entire body of `case_upper()`. This results in the large spike at the end of **Figure 1.3**.

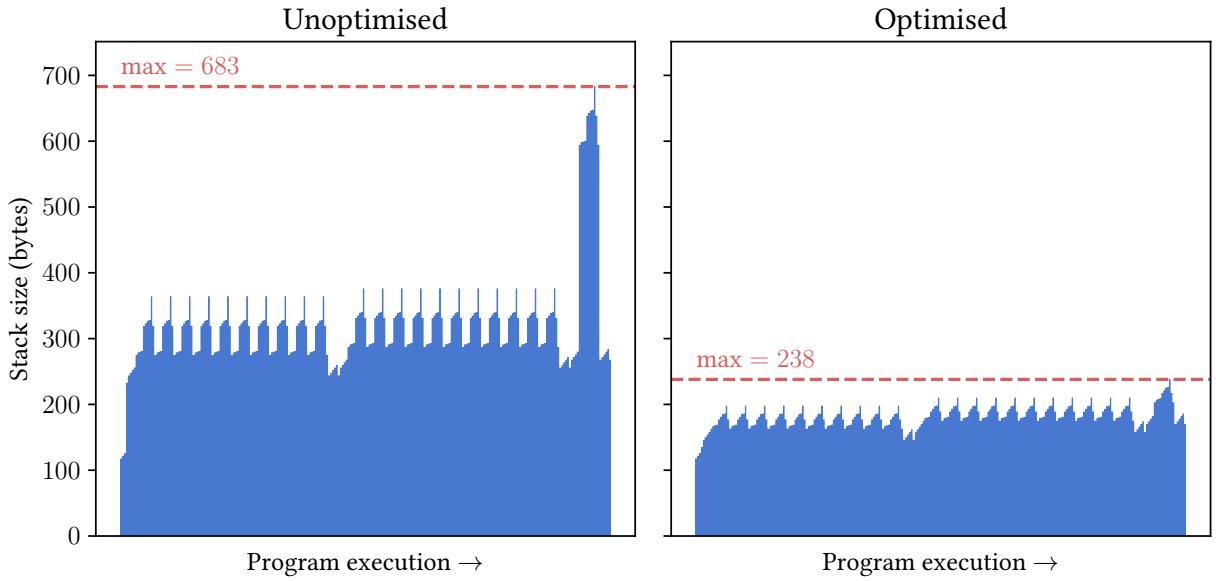

Figure 1.3: Comparing stack usage for `case.c`

```
for (char *s = str; *s; s++) {
    *s = toupper(*s);
}
return str;
```

Listing 1.2: The entire body of `case_upper()`

Due to the differences in temporary variables described above, the compiler is able to optimise `case_camel()` much more than the other functions. This parallels the fact that `case_camel()` had the largest stack frame initially.

Another area that this optimisation has a large impact is for recursive functions. Since this optimisation reduces the size of each stack frame, we will see a large improvement if we have lots of stack frames from recursion. **Figure 1.4** shows the size of the stack over the execution of calculating the Fibonacci numbers recursively. In this instance, the optimised stack allocation policy reduced the stack size of the program by 67.20 %.

1.4 Summary

```
while (*r && !CASE_IS_SEP(*r)) {
    *w = *r;
    w++;
    r++;
}
```

Listing 1.3: A short section of the body of `case_camel()`

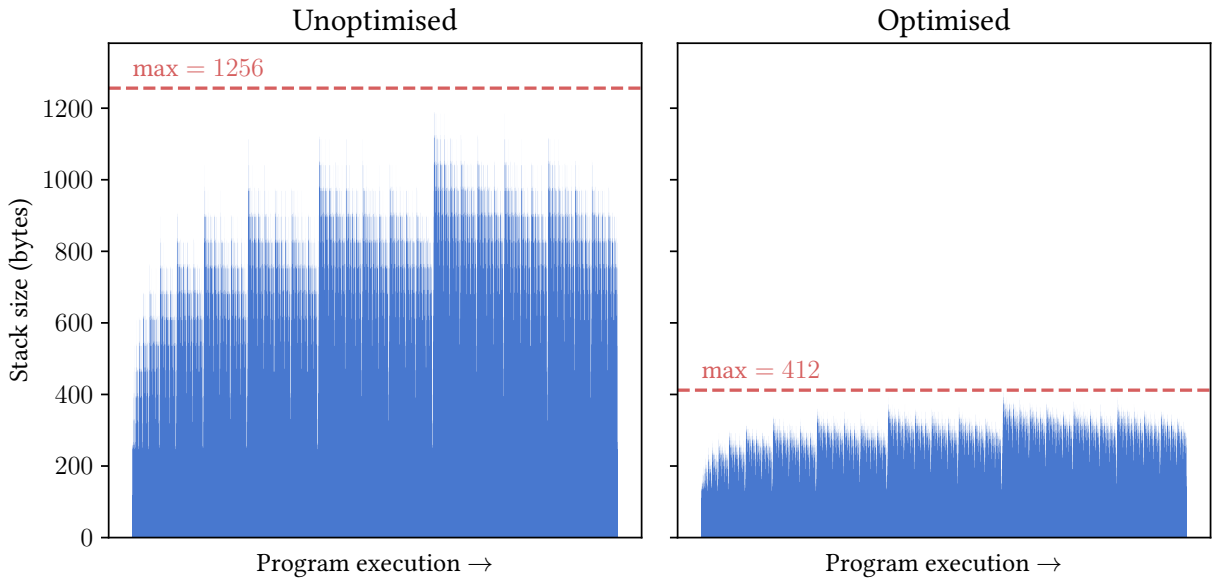

Figure 1.4: Comparing stack usage for `fibonacci.c`