1 Evaluation

1.1 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.

```
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.

1.2 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.

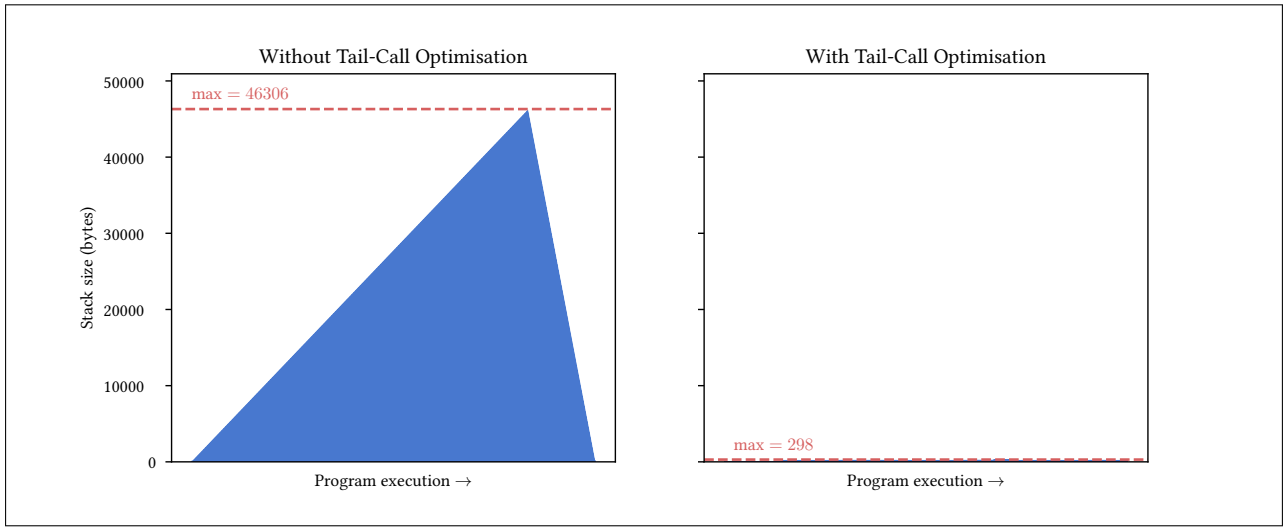

Figure 1.1: Stack usage for calling `sum(500, 0)` (see **Listing 1.1**)

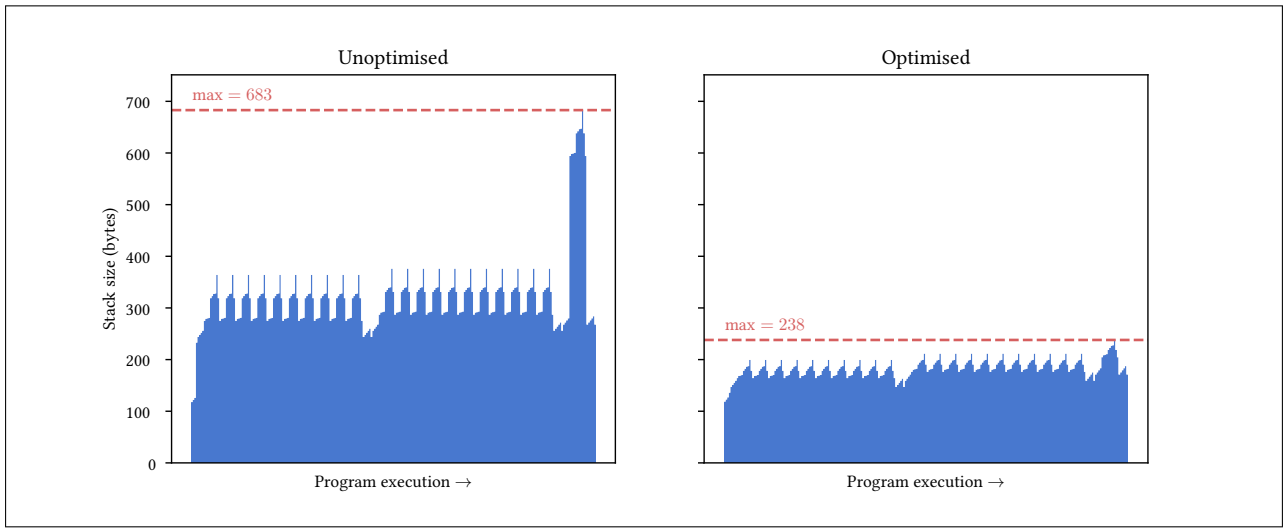

Figure 1.2: Stack profile