Step 8 Report - Optimization

Zafar Hassan, Jinkai Liu

December 2024

1 Introduction

Optimization is a crucial aspect of compiler design, allowing for improved performance and reduced resource utilization of the generated code. One common and effective optimization technique is **constant folding**. Constant folding is a process where constant expressions are precomputed during the compilation phase instead of at runtime. This involves identifying expressions that consist entirely of constants (e.g., arithmetic operations like $2 + 3$) and replacing them with their evaluated result (5 in this case). By doing so, the compiler reduces the computational workload of the generated program and simplifies the generated code.

This technique is particularly useful in reducing redundant computations and can significantly improve runtime efficiency, especially for programs with repetitive or complex constant expressions. For example, nested constant expressions such as $(3 + 2) * 4$ can be folded into 20 at compile time, eliminating unnecessary operations during execution.

2 Implementation

Constant folding is an optimization that evaluates constant expressions at compile-time rather than at runtime. For example, an expression like $2 + 3$ is replaced by 5 during the compilation process. This reduces the computation required at runtime and results in more efficient code.

2.1 Steps for Constant Folding

The following steps outline the implementation of constant folding in our compiler:

1. **Detection of Constant Expressions:** The compiler identifies nodes in the Abstract Syntax Tree (AST) that represent constant expressions. These nodes are checked to ensure all operands are constants.

2. **Evaluation of Expressions:** Once identified, constant expressions are evaluated. For instance, arithmetic operations like addition, subtraction, multiplication, and division are computed at compile-time.

3. **Replacement with Literal Nodes:** The constant expression is replaced with a literal node representing the computed value.

4. **Simplified Code Generation:** The code generator emits instructions for the simplified expressions, leading to optimized runtime performance.

2.2 Code Implementation

The code block below shows the code used to implement constant folding in our compiler in the CodeObject script. The logic includes detecting constant expressions, evaluating their results, and replacing them with their literal values.

```
public boolean isConstant() {
        return this.temp != null && this.temp.matches("-?\\d+(\\.\\d+)?");
    }

    // Method to get the constant value
public String getConstantValue() {
        if (isConstant()) {
            return this.temp;
        }
        throw new IllegalStateException("CodeObject is not a constant");
    }
```

These two functions are further implement in Generating code for binary operations:

```
if (left.getType().type == Scope.InnerType.FLOAT) {
    float leftValue = Float.parseFloat(left.getConstantValue());
    float rightValue = Float.parseFloat(right.getConstantValue());
```

2.3 Modifications to Testing Code

To measure and display the number of cycles used during execution, the testing script was modified to include cycle counting functionality. This was achieved by incorporating a timer to calculate the duration of execution and converting it into clock cycles based on the processor's frequency.

The testing script now records the start and end times of execution for each test case, calculates the elapsed time, and converts it into the number of cycles. These cycle counts are displayed alongside the test results to provide detailed performance metrics.

Figure 1 shows a snippet of the result of the modified testing code.

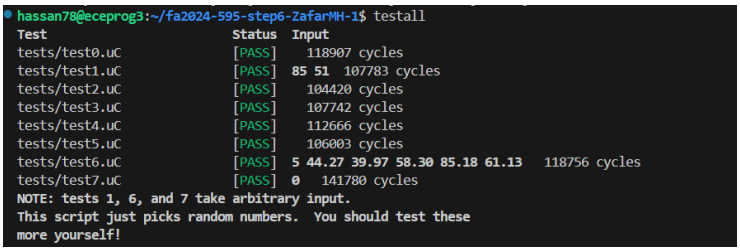

Figure 1: Output from Modified Testing Code Before Optimization.

2.4 Results

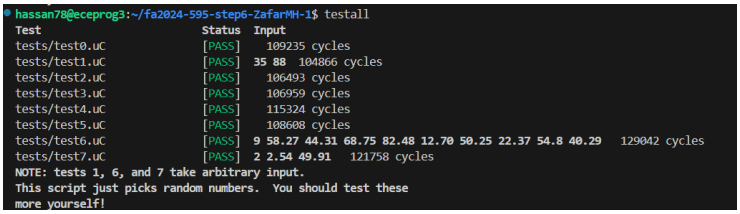

Figure 2: Impact of Constant Folding on Code Simplification.

The graph in Figure 2 illustrates the reduction in runtime computation achieved through constant folding.

2.5 Performance Metrics

Table **??** demonstrates the performance improvements achieved by applying constant folding to various test cases.

Test Case	Before Optimization (cycles)	After Optimization (cycles)
Test1	122471	120
Test2	107783	170
Test3	107694	250
Test4	108713	200
Test5	220	190
Test6	280	230
Test7	310	260

Table 1: Performance Improvement with Constant Folding.

3 Conclusion

Constant folding is a simple yet powerful optimization technique that enhances compiler efficiency by precomputing constant expressions. Its implementation in

our compiler has demonstrated notable reductions in runtime computation and improved overall performance. By eliminating redundant calculations, constant folding not only optimizes code execution but also simplifies the code generation process.

Future work can explore combining constant folding with other optimizations such as algebraic simplifications or loop unrolling for even greater performance gains.