CSCI1260 Capstone - Compiler Optimizations

Abstract

This capstone is a culmination of my various experience in Computer Systems over my time in the CS department. It focuses on optimizations that compilers implement to make our code quicker. Particularly, constant propagation & reassociation, function inlining, and peephole optimizations.

Constant propagation, Reassociation, and function inlining modify the AST itself to simplify parts of the code to make it quicker. The peephole optimization is based on assembly code where redundant instructions introduced by the compiler are modified.

Const Propagation + Reassociation

Constant Propagation is the modification of source code such that as much computation as possible is done at compile time instead of runtime. This includes replacing primitive operations (addition, subtraction, etc) with statically determined results. Additionally, let-bound names are replaced (if it is bound to a constant value) and if expressions are eliminated if possible.

Additionally, I did reassociation of the plus operator in the same pass as the constant propagation. For plus, the compiler checks for opportunities to simplify expressions such as `(+ (+ 2 (non_const_value))` which otherwise would not be simplified.

Function Inlining

Function Inlining is where static function calls are replaced by function bodies depending on some heuristic. Only static functions are inlined, so no function that calls another function (unless such function is inlined) is inlined. Before this, a pass is done to make all variable and parameter names globally unique.

My heuristic is based on two boundaries: function size and number of static call sites. Function size is calculated by counting up the number of subexpressions and number of static call sites is determined by looking at the function and program bodies. It is manually adjustable, but currently functions that are called over 8 times or over a size a 16 are not inlined.

Peephole

For the peephole optimizations, I optimized out redundant mov instructions and reassociated redundant Add/Sub commands. For Add/Sub, anything of the form “Add Reg r1, Imm x; Add Reg r1, Imm y” turns into “Add Reg r1, Imm x + y” (similar to reassociation in constant propagation). Otherwise, move instructions of the following two forms were changed: “Mov R, S; Mov S, R” -> “Mov R, S”, “Mov R, v1; Mov R, v2” -> “Mov R, v2”.

An example can be found in *mov-num-list.s,* where the optimization was applied to the “mov-num” function:

Without Peephole:

```
    mov rax, QWORD [-8 + rsp]
    mov QWORD [-16 + rsp], rax
    mov rax, QWORD [-16 + rsp]
    mov QWORD [-24 + rsp], rax
    mov rax, QWORD [-24 + rsp]
    mov QWORD [-32 + rsp], rax
    mov rax, QWORD [-32 + rsp]
```

With Peephole:

```
    mov rax, QWORD [-8 + rsp]
    mov QWORD [-16 + rsp], rax
    mov QWORD [-24 + rsp], rax
    mov QWORD [-32 + rsp], rax
```

As seen above, the peephole optimizations to this function effectively cut the instructions in half (potentially because the function was made to showcase this optimization :D).

Performance

I ran the benchmark test suite with a configuration of a base compiler with nothing, one with constant propagation, one with inlining, and one with peephole.

Speedup depended on which specific test was run. Peephole provided a speedup larger than sometimes expected - though it was clear that this was at times because it effectively replaced constant propagation.

From my inspection, each optimization did speed up tests that were designed to showcase it. There was no clear winner among the optimizations as performance varied between them all. Some statistics are below from 2/3 of the results from the benchmarks.

Number of Times Optimization was fastest (about 2/3 of our tests)

- No Optimization: 3
- Constant Propagation: 19
- Inlining: 15
- Peephole: 18