Quickly Finding RISC-V Code Quality Issues with Differential Analysis

Luís Marques
<luismarques@lowrisc.org>

LLVM Dev Meeting, Oct 2020

Problem

The Good News

- RISC-V target accepted into LLVM
  - Successfully compiled thousands of Linux packages
- Overall high performance generated code
  - Benefiting from LLVM's target-independent optimizations
  - Benchmark speed results similar to RISC-V GCC

The Bad News

- We still had cases of poor RISC-V code generation for various code patterns (e.g. simple expressions)
- These issues had gone unnoticed when we looked at the generated code for large programs and benchmarks
- How could we quickly find them?

Approach

- Project LongFruit: differential analysis of Clang vs GCC
- Python tool
  - The simplest possible implementation that could work
  - Custom random C code generator
    - Recursive descent direct code generator
    - Optimized for our needs (focuses on problematic areas)
  - RISC-V assembly parser and instruction cost estimator
    - Very simple cost model, based on instruction class (ALU, FPU, load/store, branch, etc.)
  - Plumbing to: run the random C code generator; compile the source with both Clang and GCC; analyze the resulting assembly; compare the estimated costs; filter out uninteresting cases; run a code reducer on the source code; save each reduced case to a file

Results

- Very simple tool, but highly effective
- Finds candidate issues in a few seconds, reduces them in a few minutes
- Immediately found many cases of low-hanging fruit
  - Manual triage reduced the initial batch to around a dozen independent issues.
- Code quality issues spanned a variety of categories.
- Resulted in multiple patches to address those issues
  - We still have a backlog of issues to address

Related Work

- Finding Missed Optimizations in LLVM (and other compilers). G. Barany, 2018 European LLVM Developers Meeting.

Source

```
int f(int a, int b) {
    return -(a == b);
}
```

GCC 9

```
sub     a0, a0, a1
seqz    a0, a0
neg     a0, a0
ret
```

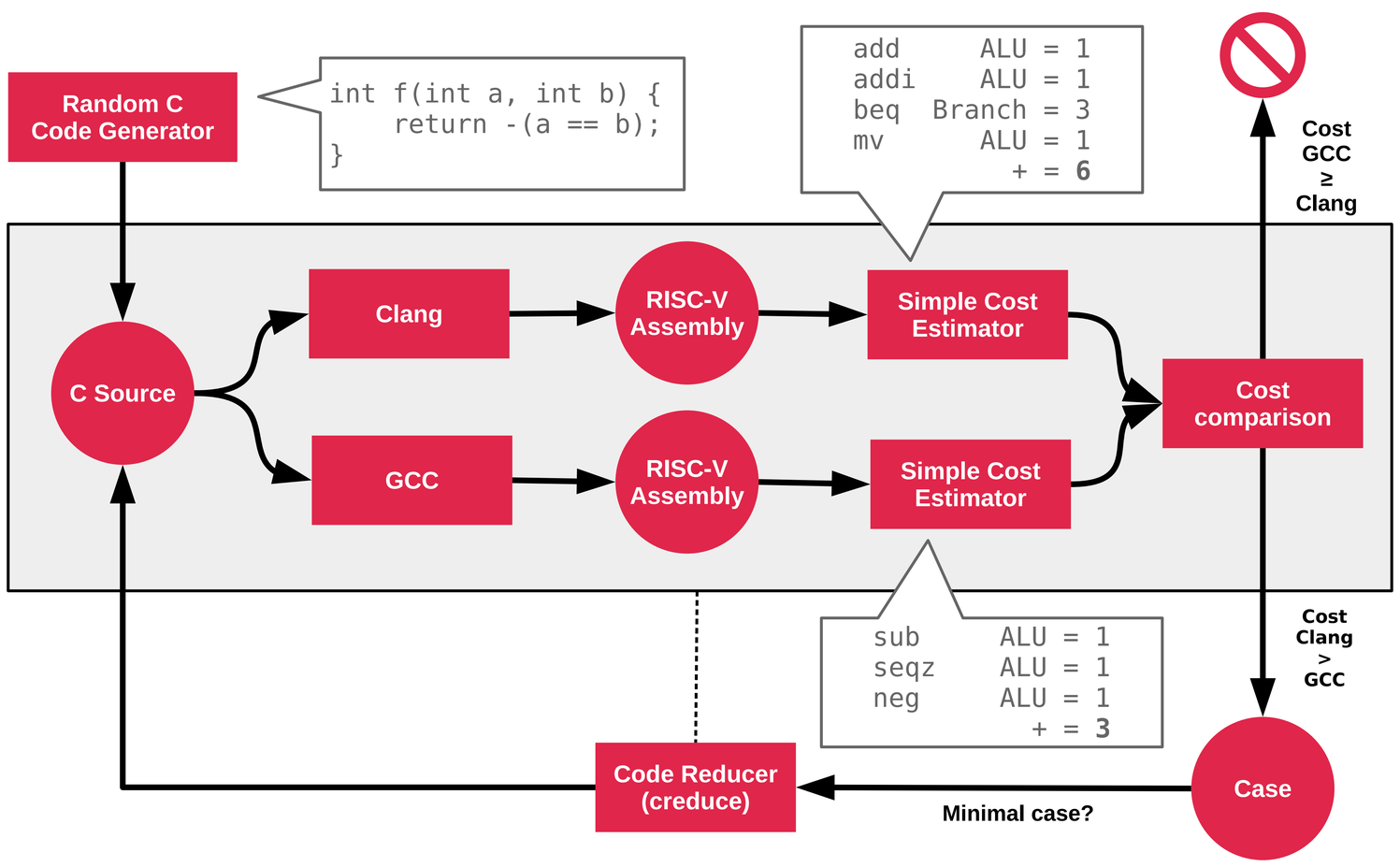
Good assembly code

How many more cases like this were still out there?

Clang 9

```
  add     a2, zero, a0
  addi    a0, zero, -1
  beq     a2, a1, .LBB0_2
  mv      a0, zero
.LBB0_2:
  ret
```

Unnecessary branch



Example

Inefficient use of offsets in loads and stores

C source

```
float f() {
    return 1.0;
}
```

Test Scenario

```
RV64GC ILP64D
```

GCC 10 Output

```
lui     a5,%hi(.LC0)
flw     fa0,%lo(.LC0)(a5)
ret
```

Clang 10 Output

```
lui     a0, %hi(.LCPI0_0)
addi    a0, a0, %lo(.LCPI0_0)
flw     fa0, 0(a0)
ret
```

Fix https://reviews.llvm.org/D79690

Clang 11 Output

```
lui     a0, %hi(.LCPI0_0)
flw     fa0, %lo(.LCPI0_0)(a0)
ret
```

Many more...

- Poor constant materializations
- Unnecessary sign extensions
- Use of branches instead of comparison instructions
- Extraneous floating-point conversions
- Dead instructions

More examples:

https://github.com/lowRISC/longfruit