

15-411 Compiler Design, Fall 2017

Lab 5

Jan and co.

Compilers Due: 11:59pm, Tuesday, November 21, 2017
Papers due 11:59pm, Tuesday, November 28, 2017

1 Introduction

The goal of the lab is to implement some optimization for the language L4, which remains unchanged from Lab 4. This includes an *unsafe* mode in which your compiler may assume that no exception will be raised during the execution of the program (except due to `assert`). This affects operations such as integer division, arithmetic shift, array access, and pointer dereference.

2 Preview of Deliverables

In this lab, you are not required to hand in any test programs, since there is no change in language specification. Instead, we will be testing your compilers on the test suites from Labs 1–4. We will also test the efficiency of your generated code on benchmark tests created by the course staff. Since you have a choice between multiple different optimizations, the benchmarks are designed as realistic programs rather than to verify that any particular optimization is taking place.

You are required to turn in a complete working compiler that translates L4 source program into correct target programs in x86-64. In addition, you have to submit a PDF file which describes and evaluates the optimizations that you implemented.

3 Compilation to Unsafe Code

The `--unsafe` flag to your compiler allows it to assume that no exceptions will be raised during the execution of the program except ones due to `assert`. This means you can eliminate some checks from the code that you generate. You are *not* required to eliminate all (or, indeed, any) checks, but it will make your compiled code slower if you do not take advantage of this opportunity at least to some extent.

Note that this does not in any way constitute an endorsement of unsafe compilation practices. It will, however, give you a more level playing field to compare the efficiency of your generated code against gcc and others in the class.

4 Optimizations

Besides unsafe mode, you are required to implement and evaluate at least **four** optimizations from the list below.

1. **Improved instruction selection.** You may optimize instruction selection to generate more compact or faster code. This includes generating good code for conditionals (e.g., avoiding `set` instructions), loops (e.g., enabling good branch prediction or aligning jump targets), and other improvements on your code.
2. **Constant propagation and folding.** Implement constant propagation together with constant folding and eliminating constant conditional branches.
3. **Dead code elimination.** Implement dead code elimination using the analysis described in the lecture notes.
4. **Eliminating register moves.** Explore techniques for eliminating register moves such as improved register allocation, copy propagation, register coalescing, and peephole optimization. We suggest coalescing registers in a single pass after register allocation as suggested by Pereira and Palsberg.
5. **Common subexpression elimination.** Implement common subexpression elimination, with or without type-based alias analysis to avoid redundant loads from memory.
6. **Loop optimizations.** Hoisting invariant computations out of loops and strength reduction enabled by basic and derived induction variables are good targets for optimizations. Another important loop optimization is loop unrolling.
7. **Tail call optimization.** Turning recursive calls at the end of functions into jumps.
8. **Inlining.** Transforming code that use function calls into code that contains a copy of the called function. For inlining, it will be important to develop a good heuristic for when to perform it.
9. **Other optimizations.** Feel free to add other optimizations as you see fit, although we strongly recommend first completing basic ones before you go for more advanced ones.

If you have already implemented any of the optimizations, you may revisit and describe them, empirically evaluate their impact, and perhaps improve them further.

In addition to the `--unsafe` flag, your compiler must take a new option, `-On` (dash capital-O), where `-O0` (dash capital-O zero) means no optimizations, `-O1` performs some optimizations, and `-O2` performs the most aggressive optimizations. One way to think about this is that `-O0` should minimize the compiler's running time, `-O2` should prioritize the emitted code's running time, and `-O1` should balance the two. (We suggest making `-O1` the default behavior.)

5 Regression Testing

As you are implementing optimizations, it is extremely important to carry out regression testing to make sure your compiler remains correct. We will call your compiler with and without the `--unsafe` flags at various levels of optimization to ascertain its continued correctness.

6 Deliverables and Deadlines

For this project, you are required to hand in a complete working compiler for L4 that produces correct target programs written in Intel x86-64 assembly language, and a description and assessment of your optimizations. The compiler must accept the flags `--unsafe` and `-On` with $n = 0, 1, \text{ or } 2$. When we grade your work, we will use the `gcc` compiler to assemble and link the code you generate into executables using the provided runtime environment on the lab machines.

Note that this time we will not just call the `_c0_main` function in the assembly file you generate, but other, internal functions in order to obtain cycle counts that are as precise as possible. So it is critical that your code follow the standard calling conventions and function naming conventions from Labs 1–4.

Compiler

The sources for your compiler should be handed in via Github as usual, and must contain documentation that is up to date. Your submissions will be tested for correctness against the full regression suite and for performance against the benchmarks. Remember that optimizations are required to preserve memory safety unless `--unsafe` is specified. You may use up to five late days for the compiler.

Compilers are due **11:59pm on Tuesday, Nov 21, 2017**.

Project Report

The project report should be a PDF file of approximately 3–5 pages, and should also be handed in on Gradescope. Your report should describe the effect of `--unsafe` as well as your optimizations and assess how well they worked in improving the code, over individual tests and the benchmark suite.

At the *absolute minimum*, your project should present a description and quantitative evaluation of the optimizations you performed at the `-O0`, `-O1`, and `-O2` levels. A good report must also discuss the way your individual optimizations *interact*, backed up by quantitative evidence. (Tables are a good idea. Graphs are an even better idea.) Make sure to carefully document how you got your numbers; someone with access to your code should, if they're willing to buy whatever hardware and operating system you were using, be able to replicate your results. A good report should also spend some time describing the effect of individual optimizations on the code you produce.

If you use algorithms that have not been covered in class, cite any relevant sources, and briefly describe how they work. If the algorithms have been covered in class, cite the appropriate lecture notes, and do not waste any space on describing the optimizations.

Project reports are due on **11:59pm on Tuesday, Nov 28, 2017**.

Late days: The late day rules for written assignments apply. You can only use late days if all team members have late days left. If you use late days then all team members will lose a late day.

Grading

This assignment is worth 150 points. 50 points will be based on your written report. 20 points will be based on the number and completeness of the optimizations you perform (as determined by reading your code and your report). Finally, 80 points will be based on the correctness of your

compiler and on the performance of your emitted code relative to our benchmarks. Details about how we grade your compiler will be announced on Piazza soon.