15-411 Compiler Design, Fall 2019
Lab 5

Seth and co.

Compilers Due: 11:59pm, Tuesday, November 19, 2019
Papers due 11:59pm, Tuesday, November 26, 2019

1 Introduction

The goal of the lab is to implement optimizations 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 compiler's optimizations on a suite of benchmark tests created by the course staff. We offer a choice of many optimizations you can implement, and the benchmarks are designed to resemble realistic programs. As in real life, some may respond better to particular optimizations than others. In addition to the benchmarks, we will be testing the correctness of your compiler by running it on the test suites from Labs 1–4, so you must be careful to maintain the semantics of the L4 language when optimizing.

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. **Conditional Constant Propagation and Folding.** Implement constant propagation together with constant folding and eliminating constant conditional branches. We recommend Sparse Conditional Constant Propagation, an SSA-based approach to this problem.

2. **Dead Code Elimination.** Implement aggressive dead code elimination using the analysis described in class. This yet another case where life is made easier by manipulating SSA-form programs.[1]

3. **Improved Register Allocation.** Explore techniques for eliminating register moves such as improved register allocation, copy propagation, register and memory coalescing, and peephole optimizations. This is likely to include heuristics such as loop depth for generating appropriate spills, and making use of all available registers when possible.

4. **Partial Redundancy Elimination.** Implement removal of partially redundant expressions using the analysis described in lecture and the related papers. This will provide the additional benefit of loop invariant code motion and common subexpression elimination.

5. **Redundant Safety Check Elimination.** Implement a dataflow-based approach to eliminating redundant null-checks and array bounds-checks when dereferencing pointers or accessing arrays at runtime. This optimization is specifically tailored towards speedup in safe mode, as these checks can be removed entirely when running with `--unsafe`.

6. **Loop Optimizations.** Implement loop unrolling, and loop fusion or loop tiling/blocking. You will require good heuristics for applying these optimizations, and we recommend consulting the lecture notes as well as the existing literature in textbooks and papers on this subject.

7. **Software Pipelining.** Possibly in combination with loop unrolling or code restructuring, implement reordering of instructions based on data-dependency analysis to allow out-of-order execution on a modern processor. This is a challenging optimization to get right!

8. **Strength Reductions and Peephole Optimizations.** Modify expressions to equivalent, cheaper ones. This includes unnecessary divisions, memory loads, and in particular redundant indexing computations that can be eliminated by analyzing derived induction variables.

9. **Tail Call Optimization.** Turning recursive calls at the end of functions into jumps, and performing accumulation transformations on tail-call expressions when required.

10. **Other optimizations.** Feel free to add other optimizations and analyses as you see fit, although we strongly recommend first completing basic ones before you go for more advanced ones. That said, please consult the course staff first to ensure that your planned optimizations are acceptable. We are likely to say yes to any nontrivial analysis.

[1] If you didn't know this, now you do.

If you have already implemented any of the optimizations, you may revisit and describe them, empirically evaluate their impact, and improve them further. In this case, your report should contain a decription of any improvements you made.

In addition to the `--unsafe` flag, your compiler must take a new option, `-O`n, where `-O0` means no optimizations, and `-O1` performs the most aggressive optimizations. One way to think about this is that `-O0` should minimize the compiler's running time, and `-O1` should prioritize the emitted code's running time. We suggest using some to decide between the two as the default behavior, if no flag is passed.

5 Testing

As you are implementing optimizations, it is extremely important to carry out regression testing to make sure your compiler remains correct. We heavily recommend that your optimizations be modular, and that correctness does *not* depend on a particular previous optimization. We will call your compiler with and without the `--unsafe` flags at various levels of optimization to ascertain its continued correctness, though your performance will be evaluated primarily through our benchmarks. To help you test your performance, you'll see some new files in the `dist` repository:

- `tests/benchmarks/`, which contain the benchmark programs.
- `timecompiler`, a script which counts the cycles of your compiler on these benchmarks.
- `gcc_bench.py`, a script to give you an idea of target cycle counts when running on your machine.
- `score_table.py`, a script which you can use to generate Notolab-like score tables.

To use the `timecompiler` script, you should follow these steps:

1. Ensure your compiler supports `--unsafe`, `-O0` and `-O1`.
2. If you wish, add additional benchmarks to the benchmark folder.
3. Run `../timecompiler --limit-run=120` from your compiler's directory. `timecompiler` accepts the same flags that gradecompiler does – however, we don't recommend running the benchmarks in parallel.

The timecompiler script has a companion: running `gcc_bench.sh` will tell you the cycle count of gcc's assembly on the benchmarks. We ran `gcc_bench.sh` on a grading instance and recorded the cycle counts of the results on all benchmarks. **A table of these is reproduced as part of the Notolab output, along with the cycle counts of your compiler's assembly**. (You can reproduce your own version of this table with score table.py.) Your target is to achieve as close to `-O1`'s cycle counts as you can, averaged over the whole benchmark suite.

Your performance will be measured as a factor of how far between `-O0` and `-O1` you are, and of course, scaled accordingly. Achieving true parity with `-O1` in the course of a single semester is a tremendous feat, and we offer the comparison primarily to give you a sense of the bigger picture.

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 `-O`n with $n = 0$, 1, 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, four other internal functions in order to obtain cycle counts that are as precise as possible. These are `_c0_init`, `_c0_prepare`, `_c0_run`, and `_c0_checksum`, each corresponding to their unprefixed counterparts in the benchmark source. Given this, it is critical that your code follow the standard calling conventions and function naming conventions from Labs 1–4 for these functions.

Compiler

The sources for your compiler should be handed in via Github as usual, and *must* contain documentation that is up to date. Particularly, your compiler should document each of your performed optimizations in both a `README` file and the source itself. The course staff *will* be reading your code as part of the submission for this lab. You may use up to five late days for the compiler.

Compilers are due **11:59pm on Tuesday, Nov 19, 2019**.

Project Report

The project report should be a PDF file of approximately 4–5 pages (possibly more, particularly with figures), and should be handed in on Gradescope. Your report should describe the effect of `--unsafe` as well as your optimizations and other improvements,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 default 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 your 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.

Your report should contain a specific commit hash for the course staff to review[2], and a comprehensive descriptions of where in your source files each optimization you have described is implemented. 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 or paper, and focus on any implementation choices you made that are not described in those resources.

Other (optional) discussions that might be included in a high-quality report include:

- Time versus space tradeoffs in emitted code.
- Effects of various optimizations on the running time of your compiler.

[2]It is not necessary that this be the same commit hash that you submit to Notolab. In fact, we encourage you to perform any code-cleanup that may be required.

- Examples of programs that your optimizations would interact particularly well with.
- Examples of programs that your optimizations would interact particularly poorly with.

Project reports are due on **11:59pm on Tuesday, Nov 26, 2019**.

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

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.

Your compiler will be graded relative to the baseline code emitted by the reference compiler and compiled via `gcc`, with `-O0` and `-O1`. We will run your compiler numerous times in all of the optimization modes, and take the k-best times of all of these. We will use the benchmark score as a multiplier for your correctness score, and we derive your multiplier by averaging your score over all of the benchmark tests. The scheme is designed so that you do not have to exceed `-O0` on all tests, and may benefit from a score greater than 1 if your optimizations provide excellent speedup in certain cases. However, you will incur a penalty of -10% for every benchmark case failed on any of the optimization modes.

The exact formula for each test is as follows: t_c is the average of the k-best times from your compiler, and t_0 and t_1 denote the times of the reference compiler using `-O0` and `-O1` respectively. The u variables denote the same times, but with `--unsafe`. $f = 0.25$ is a difficulty reduction factor chosen by the course staff, and **both P_s and P_u are clamped between 0 and $1 + f$.** T is our benchmark suite, and M is your multiplier for the lab.

$$P_s = 1 + f - \frac{t_c - t_1}{t_0 - t_1} \qquad P_u = 1 + f - \frac{u_c - u_1}{u_0 - u_1} \qquad P_{test} = \frac{P_s + P_u}{2} \qquad M = \frac{\Sigma_{t \in T}\ P_t}{|T|}$$

Your final score is then MC, where C is your correctness score from running the suites from Labs 1–4 in the standard fashion on Notolab.