15-411 Compiler Design: Lab 6 - Optimization
Fall 2010

Instructor: Andre Platzer
TAs: Anand Subramanian and Nathan Snyder

Compilers due: 11:59pm, Thursday, December 2, 2010
Term Paper due: 11:59pm, Thursday, December 09, 2010

1 Introduction

The main goal of the lab is to explore advanced aspects of compilation. This writeup describes the option of implementing optimizations; other writeups detail the option of implementing garbage collection or retargeting the compiler. The language *L4* does not change for this lab and remains the same as in Labs 4 and 5.

2 Requirements

You are required to hand in three separate items:

- The working compiler and runtime system that implement optimizing transformations.
- A performance benchmarking framework, and tests.
- A term paper describing and critically evaluating your project.

2.1 Compilers

Your compilers should treat the language *L4* as in Labs 4 and 5. While we encourage you to continue to support both safe and unsafe compilation, you may commit to one or the other compiler and terminate with exit status 1 if `l4c` is called with the unsupported switch.

If you are implementing optimization for your *L4* compiler, you have complete freedom which ones to choose. The ones discussed in lecture so far and the textbook should be considered generally important and constitute good choices. If you would like to specifically target safe compilation you may pick array bounds check elimination and optimizations based on the induction variable of a loop.

Grading criteria includes:

1. The correctness of the compiler (including the optimizations). Make sure that you do regression testing on all test cases accumulated through the semester.

2. The scope and complexity of the implemented optimization(s). You can implement several widely simple, widely applicable optimizations such as the options in the lab 5 handout. It is also acceptable to go for a few complex optimizations such as induction variable elimination and array bounds elimination, or sparse conditional constant propagation.

3. The efficiency of the compiled code (as opposed to the compiler). Compiled binaries for any of the test cases we have accumulated through the semester should be able to run in under 6 seconds. You should be able to measure (within reasonable bounds of experimental error) that the compiled code is smaller and/or faster where the optimization is applicable, and does not cause any significant performance regression where the optimization is not applicable.

4. The code quality of the optimization in terms of algorithm, readability, and modularity are considered. Asymptotic complexity matters. We don't necessarily require the fastest algorithms that use complex imperative data structures. However, you should aim for quadratic or sub-quadratic complexity in the size of the input code whenever possible.

5. The efficiency of the compiler (as opposed to the compiled code). We care about constant factors to the extent that you should be able to compile any of the test cases we have accumulated throughout the semester in under 20 seconds.

An overarching requirement is that you code should be documented. Without documentation, we may not be able to evaluate the aforementioned – therefore internal documentation of your sources may indirectly affect your grade.

2.2 Tests and Measurement Tools

An important aspect of optimization is correctly identifying opportunities for optimization and verifying that your optimizations lead to improvement in the quality of code. Therefore, you will be graded on these criteria. Opportunities for optimization may be present in common programming paradigms, inefficiencies in code as it is written, or inefficiencies inserted by your compiler as a part of reducing it from a more expressive language to a less expressive language. Obvious metrics for performance are the number of cycles taken to finish running a test, and the size of the code emitted by your compiler.

Keeping all of the mentioned factors in mind, you can use your own tools or repurpose freely available tools to measure the quality of your optimizations. The timer handed out in lab 5 is naive, but can be used as a starting point for more precise measurements. Whatever you do, document your testing methodology and its effectiveness.

Your optimizations should be applicable to realistic code – not code contrived for the purpose demonstrating one optimization or another. To this end, feel free to search through all of the test cases that we have accumulated through this semester for programs that contain commonly used data-structures and algorithms to assemble to performance test suite. You are not required to write new performance test cases. However, if you write any and use them to collect measurements, please hand them in. You may also consult the 15-122 course web page, for sample programs. In each case, make sure that it is obvious from the file name or comments whether a test case is something you newly created, or something you borrowed from previous test suites or other sources.

2.3 Term Paper

Your paper shold follow this outline.

1. Introduction. This should provide an overview of your implementation and briefly summarize the results you obtained.
2. Description of Optimizations. This section should include a concise description of the optimizing transformations you applied, as in Lab 5. Explain any change you needed to make to your compiler to facilitate the addition of these optimizations, and describe any special datastructures or algorithms you used.
3. Testing. Explain the methods you used to test your optimizations, and critically analyze the effectiveness of your testing methodology.
4. Analysis. Based on the measurements you made, give a critical analysis of how well your optimizations work, and what type of code it benefits.

The term paper will be graded. There is no hard limit on the number of pages, but we expect that you will have approximately 5-10 pages of reasonably concise and interesting analysis to present.

3 Deadlines and Deliverables

Project Proposal (due 11:59pm on Sat Nov 20)

Send an informal email to the course staff at `15411@symbolaris.com` declaring whether you elect to do this project for lab 6.

3.1 Compiler Files (due 11:59pm on Thu Dec 02)

There is no plan to automatically grade your compilers on autolab. Nevertheless, as for all labs, the files comprising the compiler should be collected in a directory `compiler/` which should contain a `Makefile`. **Important:** You should also update the `README` file and insert a roadmap to your code. This will be a helpful guide for the grader.

Issuing the shell command

```
% make l4c
```

should generate the appropriate files so that

```
% bin/l4c --safe -On <args>
% bin/l4c --unsafe -On <args>
```

will run your *L4* compiler in safe and unsafe modes, respectively. You can choose to support only one of these modes. The `-O`n flag will run optimization level n. It should accept $n = 0$, 1, or 2.

The command

```
% make clean
```

should remove all binaries, heaps, and other generated files.

All your material must be committed into `lab6opt` in the same way that you submitted your compiler in previous assignments.

3.2 Tests and Measurement Tools (due 11:59pm on Thu Dec 02)

In a directory called `bench/`, include any tests that you used for the purpose of performance measurements, and include the sources to any tools that you developed for the purpose of performance testing. If you have any of your own tools, include a brief README file explaining how to build and use your tools.

3.3 Term Paper (due 11:59 on Thu Dec 09)

Submit your term paper as a filed called `<team>-opt.pdf` via email to the course staff at `15411@symbolaris.com`

4 Notes and Hints

- Start small. If you optimize, make sure your instruction selection and register allocation are in decent shape. Improving these is definitely a form of optimization and should be documented in your term paper.
- Apply regression testing. It is very easy to get caught up in the race to faster code.
- Checkpoint frequently. A convincing term paper should compare before and after for your optimizations, as well as compare to the reference implementation. In order to do this you need to be able to run various versions of the compiler and collect statistics, so make sure you can continue to run older versions. Hand in frequently. Also, it is quite possible you may not be able to finish that last, grand optimization; having a decent prior hand-in is good insurance.
- Read the assembly code. Just looking at the assembly code that your compiler produces will give you useful insights into what you may need to optimize. You can also use the reference compiler on the fish machines to produce C code corresponding to your test cases. Then, you can use `gcc` on the C code and compare the performance.