15-411 Compiler Design: Lab 6 - Optimization
Fall 2009

Instructor: Frank Pfenning
TAs: Ruy Ley-Wild and Miguel Silva

Compilers due: 11:59pm, Thursday, December 3, 2009
Term Paper due: 11:59pm, Thursday, December 10, 2009

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 two separate items: (1) the working compiler and runtime system, and (2) a term paper describing and critically evaluating your project.

3 Benchmarks and Tests

You do not have to hand in any new benchmark files.

The benchmark file format is as in previous labs except that they must define several specific functions that will be used by the driver for timing purposes. Your compiler should not verify the presence of these functions so we can still run it on older suites for the purpose of regression testing. Your functions should work in both safe and unsafe modes.

- τ`* init(param : int);` This function creates an instance of a benchmark problem, such as an array to sort. The size of the data structure and therefore the running time of the benchmark will vary with `param`. The `init` function will not be timed. The type τ is different for different benchmarks, since the driver only passes this pointer around without examining its value.

  The result of `init` must be predictable, depending only on the random seed which the driver (but *not* your `init` function) can set by calling the C library function `srand`.

- `int prepare(data :` τ`*, param : int);` This prepares an instance of the problem for a timed run, but the time for this function itself will not be counted. For example, you might copy a given array so it can be sorted in place by the `run` function. The return value will be ignored.

- `int run(data : τ*, param : int);` This function runs an algorithm on the data structure and modifies it in some way to record the result. The time for this function call is the timing data we collect. This function should not call external library functions: we want to test the efficiency of your code generator, not that of the library functions. The return value will be ignored.

- `int checksum(data : τ*, param : int);` This function reduces your output to a checksum. The driver uses this to compare the checksum with the checksum obtained from a reference implementation in order to catch errors. In simple cases, such as function that computes an integer, the checksum can be the value itself.

The driver will call `init` once to get a (deterministic) problem instance. It will then call `prepare` followed by `run` and continue to repeat the `prepare`–`run` sequence a fixed number of times, calculating an average running time for `run`. After the last run, `checksum` is called and compared with the checksum of the reference implementation. It is important that `prepare` followed by `run` is idempotent, so that each successive run will execute exactly the same way.

The driver calls `init (1000)`; the run function for this parameter should take about 100ms with the reference implementation, although this aspect is not fine-tuned.

For ease of regression testing, the `main` function should, deterministically, call `init`, `prepare`, `run`, and then return the result of `checksum`. The first line of the file should read `#test return` n where n is that (deterministic) checksum.

4 Compilers

Your compilers should treat the language *L4* as in Labs 4 and 5, including `extern` declarations. 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 which has been discussed in lecture and you can find in Chapter 18.4 of the textbook.

Grading criteria include first the correctness of the compiler (including the optimizations) and second the scope of the implemented optimization(s). Generally, more widely applicable optimizations should be prefered. However, it is also acceptable to go for a particular complex optimization such as array bounds elimination. As a third criterion, the code quality of the optimization in terms of algorithm, readability, and modularity are considered. Finally, the efficiency of the compiler itself (as compared to the compiled code) is important only to the extent that each benchmarks should compile within a generous time limit of 60 seconds.

5 Project Requirements

On the Autolab server, the hand-in and status pages for the optimization and garbage collection projects are separated, since different drivers will be employed.

Compiler Files (due 11:59pm on Thu Dec 3)

As for all labs, the files comprising the compiler itself 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 -O2 <args>
% bin/l4c --unsafe -O2 <args>
```

will run your *L4* compiler in safe and unsafe modes, respectively. For backwards compatibility, the default is `--unsafe`. 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.

Using the Subversion Repository

The handout files for this course can be checked out from our subversion repository via

```
% svn checkout https://cvs.concert.cs.cmu.edu/15411-f09/<team>
```

where `<team>` is the name of your team. You will find materials for this lab in the `lab6opt` subdirectory. Or, if you have checked out `15-411/<team>` directory before, you can issue the command `svn update` in that directory.

After first adding (with `svn add` or `svn copy` from a previous lab) and committing your handin directory (with `svn commit`) to the repository you can hand in your tests or compiler by selecting

```
S3 - Autograde your code in svn repository
```

from the Autolab server menu. It will perform

```
% svn checkout https://cvs.concert.cs.cmu.edu/15411-f09/<team>/lab6opt/compiler
```

to obtain the files directories to autograde, depending on whether you are handing in your test files or your compiler.

If you are submitting multiple versions, please remember to commit your changes to the repository before asking the Autolab server to grade them! And please do not include an compiled files or binaries in the repository!

Term Paper (due 11:59 on Thu Dec 10)

You need to describe your implemented compiler and critically evaluate it in a term paper of about 10 pages. You may use more space if you need it. The recommended outline varies depending on your project. Submit a file `<team>-opt.pdf` via email to the instructor at `fp@cs`.

Your paper should follow this outline.

1. Introduction. This should provide an overview of your implementation and briefly summarize the results you obtained.

2. Optimizations. With a subsection for each separate pass or optimization you implemented, describe what you implemented and briefly state the rationale for your choice. Also give a brief description of the analysis algorithm itself, especially where it deviates from the description in the texbook or other reference material, and how it fits into the overall structure of your compiler.

3. Analysis. Critically evaluate the results of your optimizations. Show graphs or tables that demonstrate the speed-ups or slow-downs you obtained. This requires that you save either performance numbers or implementations (and ideally both) throughout the evolution of your compiler. Explain the results to the extent you can.

6 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. Besides the benchmarks we have a large test suite collected over several labs; use these for regression testing to make sure your compiler remains correct.

- 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 run `gcc` on the C code produced by the reference compiler for comparison.