CS 380C: Advanced Topics in Compilers
Assignment 4: Loop-Invariant Code Motion
Due: March 1st

February 24, 2016

Late submission policy: Submission can be at the most 2 days late. There will be a 10% penalty for each day after the due date (cumulative).

The goal of this assignment is to optimize loops by hoisting loop-invariant code out of the loop.

1 Loop Invariant Code Motion (LICM)

Implementation: Implement this algorithm as an LLVM loop pass.

Input: An LLVM loop L.

Output: The loop, with loop-invariant computations hoisted out of that loop and its inner loops as far as possible.

Preconditions: The loop simplification pass should have been performed on the function where the loop is present. This ensures that every loop has a preheader. It does not ensure that a "while" loop is converted into a do-while loop nested inside an IF; you should not assume that. You will also need information about natural loops and about dominators. (Another pass that can make LICM more effective is reaassociation, but that is not requred for the algorithm and is not included here; you should run it yourself before your pass). The complete `getAnalysisUsage()` method for your pass should look like this:

```
virtual void getAnalysisUsage(AnalysisUsage &AU) const {
  AU.setPreservesCFG();
  AU.addRequiredID(LoopSimplifyID); // Preheader and other transforms
  AU.addRequired<LoopInfoWrapperPass>(); // Identify natural loops
  AU.addRequired<DominatorTreeWrapperPass>(); // Need dominators for code motion
}
```

Algorithm: The goal of this algorithm is to hoist as many loop-invariant computations out of loops as possible. We focus only on register-to-register LLVM computations, i.e., those that do not read or write memory. We do not try to move out computations that have no uses within the loop: these could be moved after the loop (at all loop exits), but that requires patching up SSA form. We say

it is safe to hoist a computation out of a loop only if it is executed at least once in the original loop (when the loop is entered); this condition is checked using dominator information. The full list of criteria are given below.

```
LICM(L)
{
  // Each Loop object also gives you a preheader block for the loop.
  for (each basic block BB dominated by loop header,
                       in preorder on dominator tree) {
    if (BB is immediately within L) { // not in an inner loop or outside L
      for (each instruction I in BB) {
        if (isLoopInvariant(I) && safeToHoist(I))
          move I to pre-header basic block;
      }
    }
  }
}
```

`isLoopInvariant(I):`
An instruction is loop-invariant if both of the following are true:

1. It is one of the following LLVM instructions or instruction classes:

   `binary operator, shift, select, cast, getelementptr`.

   All other LLVM instructions are treated as not loop-invariant. In particular, you are not moving these instructions - `terminators, phi, load, store, call, invoke, malloc, free, alloca, vanext, vaarg`.

2. Every operand of the instruction is either (a) constant or (b) computed outside the loop.

`safeToHoist(I):`
An instruction is safe to hoist if either of the following is true:

1. It has no side effects (exceptions/traps). You can use `isSafeToSpeculativelyExecute()` (you can find it in `llvm/Analysis/ValueTracking.h`).

2. The basic block containing the instruction dominates all exit blocks for the loop. The exit blocks are the targets of exits from the loop, i.e., they are outside the loop.

Comments on the safety conditions:
Unlike the conditions discussed in class (and in lecture notes):

- You are using relaxed conditions for a non-excepting expression: hoisting it out of a loop (even a while loop) is fine even if it does not dominate all exits, i.e., it may lengthen some path on which it was never executed before.

- You are hoisting out an excepting expression (only) if it dominates all exit blocks. This guarantees that you will not cause a trap unless it would have been caused by the original program. However, you may potentially reorder two trapping instructions if you hoist one of them out of the loop and not the other. This means that the algorithm is safe for C, Fortran, C++, but not for Java or C#.

2 Getting Started with LLVM

Familiarize yourself with the LLVM compiler infrastructure since you will be using it for the next assignment too. You can find all the necessary documentation here:

http://llvm.org/docs/

You can use the following LLVM manuals to learn more about the compiler:

Downloading and building LLVM	http://llvm.org/docs/GettingStarted.html
LLVM command line tools	http://llvm.org/docs/CommandGuide/
LLVM IR Reference Manual	http://llvm.org/docs/LangRef.html
LLVM Programmers Manual	http://llvm.org/docs/ProgrammersManual.html
Writing an LLVM Pass	http://llvm.org/docs/WritingAnLLVMPass.html
LLVM testing infrastructure	http://llvm.org/docs/TestingGuide.html

Read these manuals very selectively, or you will spend far too much time on them!

(*You should have done this already:*) Download and build the source code for the latest release version 3.7.1 (do not get the source code from the SVN head); configure LLVM for a debug build and turn assertions on (Debug+Asserts build). You will be using the same for the next assignment.

Stampede on TACC: You can use any machine for development but you should ensure that your code can be built and run on Stampede (since we will be testing your submission on it). Use the login node on Stampede only for development - do not run or debug any executable on it. Run and debug your applications using the job scheduler. Read the user guide to learn how to submit jobs: http://www.tacc.utexas.edu/user-services/user-guides/stampede-user-guide#running

3 Implementation Guidelines

Please follow these guidelines precisely because the grading scripts will be based on it.

1. Create a new directory `LICM` in the `lib/Transforms/` folder of the LLVM source tree - `$SRC_ROOT`. All your source files, including your build script (`CMakeLists.txt`), should be within this directory. This is the source directory you will submit.

2. Register your pass as `simple-licm` (in your source code).

3. Create a `cmake` build script to build `LICM.so` in the `lib/` folder of the LLVM object tree - `$OBJ_ROOT`. You can use `$SRC_ROOT/lib/Transforms/Hello/CMakeLists.txt` as a template.

4. Append this line to `$SRC_ROOT/lib/Transforms/CMakeLists.txt`:

   ```
   add_subdirectory(LICM)
   ```

5. Run your pass using this command:

   ```
   opt -load $OBJ_ROOT/lib/LICM.so -simple-licm <$INPUT >$OUTPUT 2>$LOG
   ```

   $INPUT will be a LLVM bitcode file (`.bc`). $OUTPUT should be the optimized LLVM bitcode file. The standard error output $LOG can be used for logging information about your pass.

IMPORTANT: DO NOT READ `$SRC_ROOT/lib/Transforms/Scalar/LICM.cpp`

Try using the C++ Standard Template Library, if you haven't already - it can enable you to use effective data structures quickly (e.g., sets, maps, lists, and vectors). LLVM also offers many data structures in its library. See the LLVM Programmer's Manual for choosing an appropriate data structure. If you find you are writing code for doing something basic within LLVM, it is quite likely that the code for this exists already - use it.

You are not allowed to copy source code from anywhere, including other LLVM source files. You can include the header files and call those functions if you think it does precisely what you want. If you are not sure whether you can use something or if you think some LLVM source code is making the assignment trivial, then please let us know on Piazza (you can use a private note).

4 Testing

You are responsible for writing test cases to test your pass. We have provided one test case for your benefit. Initially, write small test cases covering all possible individual cases you can think of. We advise you to write these before you start writing your pass so that you get an understanding of what you're trying to achieve. You can later add more complex test cases. Feel free to use LLVM test programs. You should generate the LLVM bitcode file for your test cases using `-O0`; otherwise, LLVM's built-in version of LICM would have optimized your code.

You should test your pass individually as well as in combination with other passes. For example, here is a sequence of passes you can test with:

```
opt -load $OBJ_ROOT/lib/LICM.so -simplifycfg -instcombine -inline -globaldce -instcombine -simplifycfg
-adce -scalarrepl -mem2reg -adce -sccp -adce -simple-licm -verify -instcombine -dce -simplifycfg
-deadargelim -globaldce <$INPUT >$OUTPUT 2>$LOG
```

You can compare the behavior of your pass with that of LLVM's built-in version of LICM (replace `-simple-licm` with `-licm` in your `opt` command). Note that the built-in version of LICM is more complex than what you are required to implement.

We can use any program and any sequence of passes for testing your pass. Your program must be robust (no crashes or undefined behavior), precise (correct output), and optimized (faster execution time). So, please test your code thoroughly.

5 Deliverables

Submit (to canvas) an archive (preferably, `.tar.gz`/`.tgz`) of the source directory containing your source code, build script (`CMakeLists.txt`), test cases, and `README` file. Please do not submit the generated/built object files or binaries.

The `README` file should mention all the materials that you have read or used for this assignment, including LLVM documentation and source files; you can only skip mentioning header files that you have explicitly included in your source code. Mention the status of your submission, in case some

part of it was incomplete or some additional optimizations were implemented. You can also include feedback, like what was challenging and what was trivial. Please also mention your project partner.

You should also submit the test cases for which your pass worked (in the same or sub-directory); if they are publicly available, then just mention them in the README file.

Please feel free to help each other with the build system on Piazza since that is not focus of this class.

6 Grading

We will do the following on Stampede:

1. Unzip the archive you submit into the `$SRC_ROOT/lib/Transforms` folder, which will contain a `CMakeLists.txt` that adds your directory as a sub-directory.
2. Run `make` inside your sub-directory in the `$OBJ_ROOT/lib/Transforms` folder.
3. Use `opt` as mentioned in the implementation guidelines and testing to test your pass on a few input programs. Your program should not crash.
4. For simple tests, compare (`diff`) your output (converted to `.ll`) against the expected output and it should match; for more complex tests, compare the performance of your output (compiled to executable) against that of the expected output and it should perform similarly.

Note that this will be done through scripts. So, please follow the implementation guidelines precisely.

Acknowledgements

This is an adaptation of a project in Dr. Vikram Adve's Compiler Construction course at UIUC.

Any clarifications and corrections to this assignment will be posted on Piazza.