Project 5: Loop Invariant Code Motion

ECE 466/566 Fall 2015

Due: November 16, 2015

You are encouraged to comment directly on this document!

Objectives

- Implement an LLVM library for loop invariant code motion.
- Gain experience with the Loop and LoopInfo API in LLVM.
- Learn best techniques for modifying the IR to optimize a program.
- Examine technical requirements of Loop Invariant Code Motion that stem from a detailed implementation in LLVM.
- Examine the effectiveness of your implementation of LICM both on its own and in the presence of other optimizations.

Specification

In this project, you will implement Loop Invariant Code Motion (LICM) using the algorithm described in the lecture notes and then evaluate its effectiveness on wolfbench.

Here is an enhanced algorithm for LICM like the one we discussed in class:

```
LICM(LLVMLoopRef L):
      PH = LLVMGetPreheader(L)
      if PH==NULL:
            LICM_NoPreheader++
            return
      for each instruction, I, in L:
            if LLVMMakeLoopInvariant(L,I)==true:
                  LICM_Count++
                  // Handles all cases other than loads or stores
                  // Assume I gets blown away by LLVMMakeLoopInvariant
                  go to next instruction
            else:
                  if LLVMIsALoadInst(I):
                        addr = LLVMGetOperand(I,0)
                        if canMoveOutOfLoop(L,I,addr):
                              clone = LLVMCloneInstruction(I)
                              place clone at end of PH before branch
                              LLVMReplaceAllUsesWith(I,clone)
                              erase I
```

```
bool canMoveOutOfLoop(Loop L, Instruction I, LLVMValueRef addr):
     if (addr is a constant and there are no stores to addr in L):
          return true
     if (addr is an AllocaInst and no stores to addr in L and
       AllocaInst is not inside the loop):
          return true
     if (there are no stores to any address in L && addr is not
defined inside the loop && I dominates L’s exit):
          return true
     return false
```

There are several aspects of this implementation that are unspecified. Some of these issues are discussed in the following sections, but *some are left for you to discover*! Make reasonable assumptions that still let you find a lot of opportunity.

Your implementation should also count the following:

1. LICM_Count: total number of instructions moved out of the loop.
2. LICM_Loads: total number of loads moved out of the loop.
3. LICM_NoPreheader: number of loops with no pre-header.
4. LICM_BadStore: number of stores that prevent optimization in a loop.
5. LICM_BadCalls: (this is a hint!) number of call instructions that prevent optimization of a loop.

You may want to include these variables (declared as externs) in your Summary implementation and dump them to file so that you can print them using fullstats.py. Note, this will cause linker errors initially in your p3/p4 projects. You can either disable building of p3/p4 or make them link with the LICM and CSE libraries.

- To make p3 and p4 work, you should edit the CMakeLists.txt file in the p3 and p4 directory so that the libraries listed in the last line matches the last line of p5/CMakeLists.txt.
- To disable p3 and p4 (with the benefit that compiling is faster), open the projects/tools/CMakeLists.txt and remove the p3 and p4 subdirectory commands. For that matter, you can remove all but p5 if you wish.

Also, this project submission requires that you answer a few short questions, found at the end of this document. Don’t forget to include them!

Moving Instructions Out of the Loop

LLVM frowns upon moving instructions. It sounds contradictory, but it really isn’t when you understand it at a deeper level. Instead of moving an instruction from one basic block to another one, the preferred method is as follows:

1. Clone the instruction.
2. Insert the clone into the desired block.
3. Replace all uses of the old instruction with the new instruction.
4. Delete the old instruction.

This may seem more complicated than just moving an instruction. But, moving instructions requires updating a lot of state associated with basic blocks, uses, and instruction lists. It's often simpler to provide a copy, insert and then delete than to manage all of the state and ensure it's in working order.

You should use LLVMCloneInstruction to create a copy of any instruction. You should then use the Builder API to place the instruction in the preheader.

Extra Files: cfg.h and loop.h

To assist with some of the project's requirements, you'll need to use the cfg library. Please read these header/library files carefully to find functions you **will need.** You will find the library code here:

```
projects/libs/dominance
projects/libs/cfg
```

The header files are here:

```
projects/include/dominance.h
projects/include/cfg.h
projects/include/loop.h
projects/include/valmap.h
```

C projects can call these libraries directly using the provided header file. C++ projects can either call it or extract the useful code and integrate it into their library.

You will find loop.h especially useful. It has several functions for getting loops and iterating over their blocks. For example, using those functions you can iterate over all the loops.

```
#include "loop.h"
static int IterateOverLoops(LLVMValueRef F)
{
  LLVMLoopInfoRef LI = LLVMCreateLoopInfoRef(F);
  LLVMLoopRef Loop;

  for(Loop=LLVMGetFirstLoop(LI);
      Loop!=NULL;
      Loop=LLVMGetNextLoop(LI,Loop))
    {
      // Use Loop to get its basic blocks
    }
  return 0;
}
```

LLVM API

You will need to make use of the following:

```
LLVM-C/Core/Values/Modules
LLVM-C/Core/Values/BasicBlock
LLVM-C/Core/Values/Instructions
LLVM-C/Core/Values/General APIs
LLVM-C/Core/Values/User Value
LLVM-C/Core/Values/Constants/Global Values/Global Variables
projects/include/dominance.h
projects/include/cfg.h
projects/include/loop.h
```

Infrastructure Details

1. Provide an implementation for the optimizations described earlier. A starting point is provided for you in the projects directory.
    a. The code you implement should be added to the LICM library, not the p5 tool, so that it can be re-used in later projects.
    b. Your code should be added to `projects/lib/LICM/LICM_C.c` for C or `projects/lib/LICM/LICM_Cpp.cpp` for C++. A stub version of the entry point to the optimization function has already been implemented for you in each file..
    c. You may not change the name of the `LoopInvariantCodeMotion(_C/_Cpp)` function.
2. A tool has already been implemented that calls the library code and links against it. You may need to update your repository to get the latest version of code. Then rebuild everything:
    a. `cd path/to/ECE566Projects`
    b. `git pull`
        i. If this command fails, it's because you have modified files. You can either commit them or stash them (but not both). A commit will keep your

changes in the local directory, but ***stash will remove your changes*** ***and save them*** elsewhere. Pick the best one for your case:

1. `git commit -a -m"some changes I made blah blah blah"`

   Or...
2. `git stash`

Now, go back and re-execute `git pull`.

c. `cd path/to/ECE566Projects/projects/build`

d. Note: If your build directory is already configured for the language you want, then you can skip this step and go straight to building (step e). Otherwise, follow the instructions for the language you prefer. For C, remove the build directory and re-run the cmake configure command inside the build directory to choose C (and I think the default language is C++ for p2):

   i. `export LLVM_DIR=/usr/local/llvm/3.6.2/install-debug/share/llvm/cmake`

   ii. `cmake` **`-DUSE_C=1`** `-DLLVM_CMAKE_DIR=/usr/local/llvm/3.6.2/install-debug/share/llvm/cmake ..`

   For C++, remove the previous build directory and re-run the cmake configure command inside the build directory to choose C++ (I think for C++ you can skip this part):

   iii. `export LLVM_DIR=/usr/local/llvm/3.6.2/install-debug/share/llvm/cmake`

   iv. `cmake` **`-DUSE_CPP=1`** `-DLLVM_CMAKE_DIR=/usr/local/llvm/3.6.2/install-debug/share/llvm/cmake ..`

e. `cmake --build .`

3. Make a new p5-test directory in the same directory as wolfbench. Then, you can configure the project as shown below.
   a. `../wolfbench/configure` **`--enable-customtool`**``=`cd ../projects/build/tools/p5/; pwd`/p5``
   b. Then, build the test code:
      i. `make all test`
      ii. ``../wolfbench/timing.py `find . -name '*.time'` ``
   c. If you want the output to be less verbose, run it this way:
      i. `make -s all test`
   d. To validate that your optimized code produces the correct output, use the compare rule:
      i. `make compare`
   e. If you encounter a bug, you can run your tool in a debugger this way:

```
i.    make clean
ii.   make DEBUG=1
```

This will launch gdb on your tool with one of the input files. You can set breakpoints directly in your LICM implementation.

Collecting Results

The timing.py script collects the time it took for each program to run. At first, this won't be very interesting. But, as you get your project working, you can compare the performance with and without your optimizations.

By default, the p5 tool will run your LICM pass. If you want to disable it for testing, do it like this:

```
make EXTRA_SUFFIX=.no-licm CUSTOMFLAGS=-no-licm test
```

On the other hand, if you want to run register promotion and dead code elimination before your pass, you can do it like this:

```
make EXTRA_SUFFIX=.preopt+CSE CUSTOMFLAGS=-pre-optimize test
```

The CUSTOMFLAGS variable passes specific command line flags to p5. The EXTRA_SUFFIX variable changes the name of the binary. In this way, you can compile in the same directory using multiple settings without intermediate files colliding with one another.

After compiling with different settings, the timing.py script can collect the data and compare them in a tabular format.

You may also ask p5 to dump your summary information (from Project 3) using the -summary flag:

```
make CUSTOMFLAGS=-summary test
```

Questions To Answer

When you submit your code, **<u>also submit a brief report</u> that answers the following questions. Please include data to justify your claims.**

1. **How many instructions were moved using your implementation of Loop Invariant Code Motion? Report the number for all benchmarks when no other optimizations are applied and when -loop-rotate is applied. You may apply other optimizations too, but do so consistently. The only change should be -loop-rotate. Add OPTFLAGS="-loop-rotate" to the command line in order to enable the loop rotate pass. Explain/justify your results.**

2. How many load instructions were moved versus non-load instructions? How many instructions per loop on average?
3. Which programs benefited the most from your implementation of LICM in terms of execution time? Did any slow down? Use data to justify your argument.
4. Briefly explain how you calculated BadCalls and BadStores.

Getting Help

History shows that my specs are sometimes incomplete or incorrect. Therefore, please start early. If you run into problems, please post a question with a relevant hash tag on Piazza.

Grading

ECE 566 students must work individually, but ECE 466 students are allowed and *encouraged* to form groups of two. Only one student needs to submit in ECE 466, but both names must appear in the submission document and in the comments of all your code.

Uploading instructions: Upload your LICM_C.c or LICM_Cpp.cpp file.

However, if you modified other files, then please upload an archive of your entire projects folder, and add a note to your submission in the Notes field indicating which language you used. We will test your code using the test cases provided in wolfbench and with some secret cases we did not provide.

The assignment is out of 100 points total. If you make no attempt and submit the provided code without meaningful changes (i.e. white space and comments do not count), you earn 10 points. Otherwise, assigned as follows:

ECE 466
10 points: Compiles properly with no additional warnings or errors, and produces no verbose output *other than required counters*.
05 points: Code is well commented and written in a professional coding style
30 points: Meets or exceeds all specifications.
40 points: Optimizes all test cases.
15 points: Report, points divided evenly per question.

ECE 566
10 points: Compiles properly with no additional warnings or errors, and produces no verbose output *other than required counters.*
05 points: Code is well commented and written in a professional coding style
30 points: Meets or exceeds all specifications
30 points: Optimizes all test cases.
10 points: Optimizes secret cases.

15 points: Report, points divided evenly per question