**Bonus Project 1: Global Code Motion**

ECE 466/566 Fall 2014

Due: December 17, Noon

This is an individual project for all students in 466 and 566.

Worth up to 80 Bonus Points on your project grade.

Your total project grade may not exceed 100 points

*You are encouraged to comment directly on this document!*

**Objectives**

* Implement an optimization from a high-level specification.
* Gain experience with the User, Operands, and Dominance API in LLVM.
* Learn best techniques for modifying the IR to optimize a program.
* Examine technical requirements of Global Code Motion that stem from a detailed implementation in LLVM.
* Examine the effectiveness of your implementation of GCM both on its own and in the presence of other optimizations.

**Specification**

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

Here is the algorithm we discussed in class:

GCMOnInstruction(Instruction I):

BasicBlock Early = Nearest Dom to I for operands of I

BasicBlock Late = LeastCommonAncestorInDomTree(Uses of Inst)

BasicBlock Best = Late

while Late != Early:

Late = ImmediateDominator(Late)

if LoopNest(Late) < LoopNest(Best):

Best = Late

Place I in Best

You should make your code match this implementation as closely as possible. However*, there are several aspects of this implementation that need clarification when implemented*. Several of these issues are discussed in the following sections, but *some are left for you to discover*!

Please print the following values:

* GCM\_Count: number of moved instructions
* GCM\_AvgDistance: average distance in basic blocks an instruction is moved.
* GCM\_AfterLoop: count the number of instructions that are moved to a block that is after the loop it initially started in.
* GCM\_BeforeLoop: count the number of instructions moved before the loop the instruction initially started in.

**Not Everything Should Be Moved**

Some instructions cannot be moved using this algorithm. In some cases, namely loads, stores, and calls, it’s because we haven’t proved all the conditions required for correctness. In others, namely phi-nodes and branches, the operation is required for the correctness of the IR.

In total, the following instructions should not be considered by the GCM pass:

* Load
* Store
* Call/invoke
* Phi
* Branches (of any kind)
* Allocas (for simplicity)

**PHI Nodes and Least Common Ancestor**

When implementing the LeastCommonAncestor algorithm, you will encounter uses of an instruction inside a phi-node. You should not ignore these uses, however, it is also incorrect to use the location of the phi-node in your LeastCommonAncestor algorithm. Instead, you need to find the block from which the use flows to the phi-node and use that block instead.

**Moving Instructions**

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 insure it’s in working order.

**Order of Traversal**

The order of instruction traversal matters since the placement of an instruction is determined by the location of its operands and its uses. If its operands or uses are moved elsewhere, it may change an instruction’s recommended placement.

In this project, you may traverse instructions in an arbitrary order of your choosing, but only visit each instruction once. In the report you submit, you must state the order you implemented. For example, you may say: basic blocks were visited in the order given by the basic block iterator, and instructions were visited in the order specified by the instruction iterator.

**Extra Files: Dominance and CFG**

To assist with some of the project’s requirements, I’ve added two C libraries. One from p3 that provides dominator information has been extended with new functionality. I’ve also added a cfg library with some helpful functions. 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

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.

**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

**Implementation Details**

1. Provide an implementation for the optimizations described earlier. A starting point is provided for you in the projects directory.
   1. The code you implement should be added to the GCM library, not the b1 tool, so that it can be re-used in later projects.
   2. Your code should be added to projects/lib/GCM/GCM\_C.c for C or projects/lib/GCM/GCM\_Cpp.cpp for C++. A stub version of the entry point to the optimization function has already been implemented for you in each file..
   3. You may not change the name of the GlobalCodeMotion(\_C/\_Cpp) function.
2. A tool has already been implemented that calls the library code and links against it. To get the working tool and library stub, do the following:
   1. If you already have a working projects directory, all you need to do is this:
      1. git commit -a -m”save current work”
      2. git pull
      3. From the projects directory:
         1. make && make install
   2. If you are checking out a clean projects directory:
      1. cd ECE566Projects/projects
      2. git pull
      3. On VCL: *(on your linux install, remember to adjust the paths below)*
         1. ./configure --disable-optimized --with-llvmsrc=/usr/ece566/llvm-3.4 --with-llvmobj=/usr/ece566/llvm-build --prefix=`pwd`/install
      4. On a Mac:
         1. ./configure --disable-optimized --with-llvmsrc=`cd ..; pwd`/llvm-3.4 --with-llvmobj=`cd ..; pwd`/llvm-build --prefix=`pwd`/install --target=x86\_64
      5. make && make install
   3. *For faster compile time*: If you wish to disable building earlier projects, then remove all other directory names from the DIRS variable other than the one you are working on.
   4. **For C++**. Both C and C++ implementations are called from the same tool in this project. But, it is configured for C by default. **To use C++, you must do this:**
      1. Open projects/tools/b1/main.cpp with a text editor
      2. Near the top of the file, comment out: #define UseC
3. Your project will be tested using the wolfbench repository configured using the tool you implement. Configure your testing directory as follows:
   1. In the parent directory for your wolfbench directory, make a testing directory:
      1. mkdir b1-tests; cd b1-tests
      2. Assuming that wolfbench, b1-tests, and projects are all in the same parent directory, you can configure the project like this:
         1. For VCL:
            1. ../wolfbench/configure **--enable-customtool**=`cd ../projects/install/bin; pwd`/**b1** --enable-gcc=/usr/ece566/install/bin/gcc --enable-dragonegg=/usr/ece566/dragonegg.so
         2. For a Mac (make sure the ece566 install path is found first in the PATH variable before the host version of clang):

../wolfbench/configure --enable-customtool=`cd ../projects/install/bin; pwd`/**b1**

* + 1. Then, build the test code:
       1. make all && make test
       2. ../wolfbench/timing.py `find . -name ‘\*.time’`

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 b1 tool will run your GCM pass. If you want to disable it for testing, do it like this:

make EXTRA\_SUFFIX=.no-gcm CUSTOMFLAGS=-no-gcm 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 b1. 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 b1 to dump your summary information (from Project 3) using the -summary flag:

make CUSTOMFLAGS=-summary test

**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**

All students must work individually.

**Uploading instructions:** Make a backup of your source tree.

cp –R ./projects/ bonus1

Then do this:

cd bonus1

rm –Rf `find . –name Debug+Asserts`

rm [any other test files you created]

Then, zip up the folder using either zip or tar+gzip to create a single compressed archive.

tar czf bonus1.tgz ./bonus1

or

zip –r bonus1 ./bonus1

Upload the archive in Moodle on Bonus 1 assignment page. Also, please 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, you earn 0 points. Otherwise, assigned as follows:

**ECE 466/566**

+60 points: Optimizes all benchmarks.

+20 points: Describe any extensions you make to this optimization to achieve more benefit or evaluate its performance with respect to other passes. Include them in a brief report.