Project 4: Common Subexpression Elimination Plus Simple Load/Store Optimization

ECE 466/566 Fall 2015

Due: November 2, 11:59 pm

You are encouraged to comment directly on this document!

Objectives

- Implement an LLVM library that performs Common Subexpression Elimination.
- Leverage LLVM to perform simple instruction simplification.
- Perform a simple load-store optimization during the CSE traversal to find additional redundancy while preserving the order of memory operations.

Specification

In this project, you will implement Common Subexpression Elimination plus a few other simple optimizations. I'll describe each of them below.

- Common Subexpression Elimination
  - For each instruction, eliminate all other instructions which are literal matches:
    - Same opcode
    - Same type (LLVMTypeOf of the instruction not its operands)
    - Same number of operands
    - Same operands in the same order (no commutativity)
  - Do not consider Loads, Stores, Terminators, VAArg, Calls, Allocas, and FCmps for elimination.
  - To avoid implementation complexity, I recommend using a nested loop (it may actually involve recursion on the dominator tree) like the pseudocode below to implement CSE:
    - `for each instruction: i`
      - `visit all instructions j that are dominated by i`
        - `if i and j are common subexpression`
          - `replace all uses of j with i`
          - `erase j`
          - `CSE_Basic++`
  - Create a counter of all instructions you eliminate here and call it CSE_Basic. You will dump this to standard output at the end of your pass.
- Optimization 0: Eliminate dead instructions
  - While you visit each instruction, check if it is dead.
  - If so, remove the instruction.

    - Count these simplifications as CSE_Dead.
- Optimization 1: Simplify Instructions
    - While you visit each instruction, check if it can be simplified through simple constant folding.
    - Use the InstructionSimplify function that I've provided in transform.h. For C++, call the SimplifyInstruction function directly.
    - Count these simplifications as CSE_Simplify.
- (**566 only**) Optimization 2: Eliminate Redundant Loads
    - While traversing the instructions in a single basic block, if you come across a load we will look to see if there are redundant loads within the same basic block only.
    - You may only eliminate a later load as redundant if the later load is not volatile, it loads the same address, it loads the same type operand, and there are no intervening stores or calls to **any** address.
    - Here is the pseudocode you should follow:
        - `for each load, L:`
            - `for each instruction, R, that follows L in its basic block:`
                - `if R is load && R is not volatile and R's load address is the same as L && TypeOf(R)==TypeOf(L):`
                    - `Replace all uses of R with L`
                    - `Erase R`
                    - `CSE_RLoad++`
                - `if R is a store:`
                    - `break (stop considering load L, move on)`
- (**566 only**) Optimization 3: Eliminate Redundant Stores (and Loads)
    - If you find two stores to the same address with no intervening loads and the earlier store is not volatile, you should remove the earlier one.
        - These are simply back-to-back stores and the earlier one is redundant with the later one.
    - When you find a store, look to see if there is a non-volatile load to the same address after the store within the same basic block.
        - If such a case occurs, replace all uses of the load with the store's data operand.
    - Here is the pseudocode you should follow:
        - `for each Store, S:`
            - `for each instruction, R, that follows S in its basic block:`
                - `if R is a load && R is not volatile and R's load address is the same as S and TypeOf(R)==TypeOf(S's value operand):`

        - Replace all uses of R with S's data operand
        - Erase R
        - CSE_Store2Load++
        - continue to next instruction
    - if R is a store && R is storing to the same address && S is not volatile && R and S value operands are the same type:
        - Erase S
        - break (and move to next Store)
        - CSE_RStore++
    - if R is a load or a store:
        - break (and move to next Store)
- You should treat call instructions as stores to an unknown and possibly the same address.
- At the end of your pass, print a total of all instructions removed and show the breakdown across each category. You may print this directly to standard output, but please format the data in an easy to read manner.
- You may enhance my approach as you see fit. But, do not change the meaning of the counted values. If you can remove more loads/stores, then add a separate counter to track that.

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 CSE library, not the p4 tool, so that it can be re-used in later projects.
    b. Your code should be added to `projects/lib/CSE/CSE_C.c` for C or `projects/lib/CSE/CSE_Cpp.cpp` for C++. A stub version of the entry point to the optimization (LLVMCommonSubexpressionElimination) function has already been implemented for you in each file.
    c. You may not change the name of the `LLVMCommonSubexpressionElimination[_Cpp]` function. The `static` functions within the files can be changed as you see fit. You can add, remove, or modify them in any way you choose. If my comments don't make sense to you, then do it your own way.
2. 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 p4-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/p4/; pwd` ``/p4`
   b. Then, build the test code:
      i. `make all test`
   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 CSE implementation.

Analyze Your 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 p4 tool will run your CSE pass. If you want to disable it for testing, do it like this:

```
make EXTRA_SUFFIX=.no-cse CUSTOMFLAGS="-no-cse" test
```

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

```
make EXTRA_SUFFIX=.mem2reg+CSE CUSTOMFLAGS="-mem2reg" test
```

The CUSTOMFLAGS variable passes specific command line flags to p4. 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 p4 to dump your summary information (from Project 3) using the -summary flag:

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

You can pass multiple flags to your tool by putting them in quotes:

```
make CUSTOMFLAGS="-mem2reg -summary" test
```

Hints and Tips

Extra Files: Computing Dominance, Simplify Transform

To assist with some of the project's requirements, I've added a C library that provides dominator information and access to `SimplifyInstruction`. You will find it here:

```
projects/libs/dominance
projects/libs/transform
```

The header files are here:

```
projects/include/dominance.h
projects/include/transform.h
```

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

To include one of these headers in a library or tool, all you need to do is include it like this:

```
#include "dominance.h"
```

The makefiles already provide the necessary search paths during compiler-compile time so you don't need to specify the full relative path in the include directive.

Visiting Dominator Tree Children

Inside the dominator.h header, there are a few methods for traversing the dominator tree children of a basic block. It looks something like this:

```
LLVMBasicBlockRef child = LLVMFirstDomChild(BB);
while (child) {
   child = LLVMNextDomChild(BB,child);  // get next child of BB
}
```

This can be extended to perform a full traversal of the dominator tree using a recursive call on each child.

Visiting and *Removing* Instructions in a Module

You can iterate over the instructions in each function in the module as shown below. But, deleting an instruction requires some care since it effectively deletes your iterator. Shown in the code below is an idiom you can use to safely remove instructions:

```
LLVMValueRef  fn_iter; // iterator
for (fn_iter = LLVMGetFirstFunction(Module); fn_iter!=NULL;
     fn_iter = LLVMGetNextFunction(fn_iter))
{
     // fn_iter points to a function
     LLVMBasicBlockRef bb_iter; /* points to each basic block
                                   one at a time */
     for (bb_iter = LLVMGetFirstBasicBlock(fn_iter);
          bb_iter != NULL; bb_iter = LLVMGetNextBasicBlock(bb_iter))
     {
```

```
        LLVMValueRef inst_iter = LLVMGetFirstInstruction(bb_iter);
        while(inst_iter != NULL)
        {
             if (/* should remove inst_iter */) {
                LLVMValueRef rm=inst_iter;
                // update iterator first, before erasing
                inst_iter = LLVMGetNextInstruction(inst_iter);

                LLVMInstructionEraseFromParent(rm);
                continue;
             }
           inst_iter = LLVMGetNextInstruction(inst_iter)
        }
     }
}
```

Analyzing Instructions

To determine the opcode of an instruction, use either `LLVMIsA*` or `LLVMGetInstructionOpcode`.
Using this call, you can differentiate loads, stores, branches, and call instructions.

To examine an operand to an instruction, use `LLVMGetNumOperands (LLVMValueRef)` to determine how many operands it has. It's return value will tell you how many there are. Then, you can obtain a pointer to a specific operand using `LLVMGetOpearand (LLVMValueRef, int pos)`.

Since the destination block of a branch is just an operand to the branch instruction, you can find back edges by getting the operand of a branch and testing if the destination dominates the branch's block.

Volatile Memory Operations

To determine if a memory operation is volatile, use:
`LLVMBool LLVMGetVolatile(LLVMValueRef).`

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 CSE_C.c or CSE_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.

Questions: You must submit a brief report along with your code with the following details:

1. Collect data that compares the cumulative number of instructions, the number of loads, the number of stores, the counters you collect in the CSE pass, and the execution time of your pass both with and without the -mem2reg pass running first. Include this data in either a graph or table in your report.
2. Explain the difference in results for these two configurations using your data for Q1.
3. Compare the output counters you collected. What trends do you notice across the applications for CSE_Basic, CSE_Dead, and CSE_Simplify? ECE 566 students should also include CSE_RLoad, CSE_RStore, and CSE_Store2Load in their answer. Please include data to justify your answer.

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 566
10 points: Compiles properly with no warnings or errors
10 points: Code is well commented and written in a professional coding style
30 points: Meets or exceeds all specifications
25 points: Fraction of tests that pass
10 points: Fraction of secret tests that pass (these may overlap with provided tests)
15 points: Report with answers to the questions.

ECE 466
20 points: Compiles properly with no warnings or errors
10 points: Code is well commented and written in a professional coding style
30 points: Meets or exceeds all specifications
25 points: Fraction of provided tests that pass
15 points: Report with answers to the questions.