Project 2: Common Subexpression Elimination Plus Simple Load/Store Optimization

ECE 466/566 Spring 2024

Due: Thursday, March 21, 11:59 pm

You are encouraged to comment directly on this document!

Objectives

- Implement code that performs Common Subexpression Elimination.
- Leverage LLVM to perform instruction simplification and dead code elimination.
- ECE 566: Perform a simple load and 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.

- Optimization 0: Eliminate dead instructions
  - While you visit each instruction to perform CSE, check if it is dead.
  - If so, remove the instruction and move to the next one.
  - Create a counter of all instructions you eliminate and call it CSEDead.
  - **Important: this is not a dead code elimination pass. DO NOT BUFFER INSTRUCTIONS IN A WORKLIST AND REMOVE THEM LATER. The point of this optimization is to remove dead instructions if you find them along the way, and your code should not search aggressively for all dead instructions.**
- Optimization 1: Simplify Instructions
  - While you visit each instruction, check if it can be simplified through simple constant folding. This is similar in concept to a simple GVN.
  - Use the InstructionSimplify function that I've provided in transform.h. For C++, call the simplifyInstruction function directly.
  - If simplifyInstruction returns a non-null value, then replace all uses of the original instruction with the returned value and erase the original instruction.
    - Example: simplify instruction, `i` to `v`
    -
    - ```
      al:
      Value *val =
      simplifyInstruction(&*i,M->getDataLayout());
      ```
  - To replace all uses of one operand with another, use Value::replaceAllUsesWith or LLVMReplaceAllUsesWith in the C API. Example, replace `i` with `val`:
    - ```
      i->replaceAllUsesWith(val);
      ```
  - Create a counter and increment for all such instructions. Call it CSESimplify.
- 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)
    - You will need to decide which opcodes can be eliminated by CSE.
    - 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 CSEBasic. You will dump this to standard output at the end of your pass.
- (**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 of 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`
                    - `CSERLoad++`
                - `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 load 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`
        - `CSEStore2Load++`
        - `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`
        - `CSERStore++`
        - `break (and move to next Store)`
      - `if R is a load or a store (or any instruction with a side-effect):`
        - `break (and move to next Store)`
- You should treat **`call`** instructions as **stores to an unknown and possibly same address** as S or L.
- You must keep the code that will print a total of all instructions removed and show the breakdown across each category. The code to do this is already provided for you and your final submission must contain this support.
- You may enhance the CSE algorithm to make it more effective. But, your extensions must be enabled all of the time. If you do so, please add a description to your final report and justify that it helps by collecting appropriate statistics. Do not change the meaning of the counted values. If you can remove more instructions/loads/stores, then add a separate counter to track that.

Infrastructure

Save pending work

You will need to update your repository to get the latest version of code. Then rebuild everything. Either commit or stash any pending work:

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

Case 1: Basic cloned repository

Pull the latest copy and then merge conflicts:

```
git pull
```

Case 2: Remote tracking branch (Advanced option from Project 0)

If you have the advanced setup from Project 0 with a remote tracking branch, do this:

```
git checkout work # replace work with your branch of choice
```

Replay your commits on top of the class repo:

```
git pull --rebase ece566 main
```

Code Development and Testing Setup

You may implement your project in either C or C++. Use one of these files to implement and test your code:

	Stub implementation to edit	Binary to use for testing
C++	projects/p2/C++/p2.cpp	build/p2
C	projects/p2/C/cse.c	build/p2

This is only a starting point! A stub version of the entry point to the optimization (CommonSubexpressionElimination) function has already been implemented for you in either C or C++ starter file.

You may not change the name of the `CommonSubexpressionElimination` function. You may add or modify other content of the files except for how the statistics are dumped. These will be used by the grader script and must be retained as is. However, you may add more statistics if it's helpful.

CLion Setup

Import the project into CLion and configure it to use the LLVM toolchain you set up in Project 0. Repeat the steps in that document if you don't remember all of the steps. Then, build your code as usual for CLion.

Manual Setup for VCL or in the container

Make a build directory in the source folder, configure it using cmake, and then build and test your code. Note, if you are working inside the container, you will need to go to the host shared volume under /ece566. Please adjust the PREFIX path appropriately for your use case.

C	`cd PREFIX/ncstate_ece566_spring2023/projects/p2/C` `mkdir build`

	`cd build`
	`cmake ..`
	`make`

C++	`cd PREFIX/ncstate_ece566_spring2023/projects/p2/C++`
	`mkdir build`
	`cd build`
	`cmake ..`
	`make`

Testing

Test your code using wolfbench as you did in Project 0.

1. Make a directory for testing. In the `p2/C` or `p2/C++` is fine, but you can put it anywhere you want:
   a. `mkdir p2-test`
   b. `cd p2-test`

2. Assuming that `p2-test` is in your C or C++ directory:
   `path/to/wolfbench/configure`
   `--enable-customtool=/path/to/p2`

4. Compile the benchmarks to test your code:
   `make all test`
5. To verify that the test cases continue to function properly after optimization, use an extra make flag that will compare the outputs to the correct outputs:
   `make test compare`
6. If you encounter a bug, you can run your tool in a debugger this way:
   1. `make clean`
   2. `make DEBUG=1`
   This will launch lldb using your tool on the first un-compiled input file, which is likely the one failing. You can set breakpoints directly in your source files. This option will only work on the VCL image for debugging with lldb installed.
   For more info on how to use lldb.

Analyze Your Results

As part of this project, you will need to submit a report that describes how well your pass works. To accomplish this, you will need to:

1. Compile with multiple settings. For example, compile once with your pass, and once without it. You may also want to vary the passes that are performed with it, like whether

or not memory to register promotion is applied to the code or not.
2. Collect how many instructions are eliminated by your pass in various conditions.
3. Measure the performance of benchmarks with and without your pass.

To help you, I've provided scripts to help you analyze the timing and collect basic statistics about your pass.

Use Makefiles to Compile With and Without Your Pass

By default, the p2 tool will run your CSE pass. If you want to disable it for testing, do it like this:

```
make EXTRA_SUFFIX=.NOCSE 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=.M2RCSE CUSTOMFLAGS="-mem2reg" test
```

The CUSTOMFLAGS variable passes specific command line flags to p2. 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.

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

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

If you find you need to fix a bug and re-run to collect new results, make sure you run a clean:

```
make CUSTOMFLAGS="-mem2reg -verbose" clean
```

WARNING: **make clean will delete all of the previous results**, so collect them and copy them out first if you want to preserve them.

Use Scripts to Compare Statistics and Results

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

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.

Fullstats.py aggregates statistics information, and timing.py aggregates performance information.

After running and compiling the benchmarks, you can compare statistics across runs and timing using the python scripts available in wolfbench. Here's a rough template how to use them:

- `path/to/wolfbench/fullstats.py Instructions`
- `path/to/wolfbench/timing.py`

For example, if you run the two make commands as follows without any implementation of CSE:

```
make EXTRA_SUFFIX=.NOCSE CUSTOMFLAGS=”-no-cse -verbose” test
make EXTRA_SUFFIX=.M2RCSE CUSTOMFLAGS=”-mem2reg -verbose” test
```

When they are done, you run the stats command to see the change in instructions across compile settings:

```
/ece566/wolfbench/fullstats.py Instructions
Category                M2RCSE     NOCSE
adpcm.....................248.......509
arm.......................469.......967
basicmath.................368.......681
bh........................2192......4162
bitcount..................450.......774
crc32.....................91........176
dijkstra..................237.......342
em3d......................689......1480
fft.......................466.......862
hanoi.....................56........112
hello.....................2.........4
kmp.......................384.......620
l2lat.....................65........137
patricia..................763......1217
qsort.....................118.......183
sha.......................441.......733
smatrix...................236.......344
sql.....................117954....204959
susan.....................7843.....13232
```

You can pass a name of a counter on the command line to extract that particular value, such as CSEDead, CSEElim, CSESimplify, etc. Just make sure the string matches exactly how it is written in the code.

Use the timing command to see the execution time of different compilation settings. Some of the workloads do not run long enough to observe a meaningful runtime, and they show up below as 0 in both columns.

```
../wolfbench/timing.py
Category               .M2RCSE    .NOCSE
adpcm.....................0.0.......0.0
arm.......................0.0.......0.0
basicmath.................0.06......0.03
bh........................0.49......0.79
bitcount..................0.08......0.14
crc32.....................0.0.......0.0
dijkstra..................0.05......0.06
em3d......................0.45......0.47
fft.......................0.06......0.06
```

```
hanoi.....................1.67.......1.5
kmp.......................0.08......0.11
l2lat.....................0.02......0.03
patricia..................0.07......0.07
qsort.....................0.05......0.03
sha.......................0.01......0.01
smatrix...................2.68......2.74
sql.......................0.0.......0.0
susan.....................0.22......0.57
```

Hints and Tips for the C API

Note, this following discussion mainly applies to the C API. However, the C API is implemented in terms of a C++ API underneath. So C++ coders can read the C++ code to see how the C API works. Hence, this should be helpful to them as well.

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:

Directory: project/2/C
`dominance.h, dominance.cpp`
`transform.h, transform.cpp`

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

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

LLVM has an API for creating and using a dominator tree.

```
DT = new DominatorTreeBase<BasicBlock, false>(); // make a new one
DT->recalculate(F); // calculate for a new function F

DomTreeNodeBase<BasicBlock> *Node = DT->getNode(BB); // get node for BB
for (DomTreeNodeBase<BasicBlock> ** child = Node->begin(); child !=
```

```
Node->end(); child++)
{
   // iterate over each child of BB
   BasicBlock *bb = (*child)->getBlock();
}
```

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 `LLVMGetOperand (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);`

Dumping Statistics

LLVM has good support for collecting and dumping statistics. All you need to do is declare a statistics variable and increment it as needed. The name you give as the first argument to LLVMStatisticsCreate will appear in the csv file.

```
LLVMStatisticsRef CSEDead;

void CommonSubexpressionElimination(LLVMModuleRef Module)
{
    CSEDead = LLVMStatisticsCreate("CSEDead", "CSE found dead
instructions");

    // Increment CSEDead one time
    LLVMStatisticsInc(CSEDead);
```

```
}
```

You can use the fullstats.py script to collect it across all programs:
`../wolfbench/fullstats.py mystat`

Getting Help

History shows that my specs are sometimes incomplete, incorrect, or confusing. Therefore, please start early. If you run into problems, please post a question to the Google forum, but remember do not post your code unless it's a question about specific starter code that I gave you.

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 headers of all submitted files.

Questions

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

1. Describe your implementation and any embellishments you made over my description.
2. Collect data per benchmark that compares the number of instructions, the total number of loads, the total 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.
3. Explain the difference in results for these two configurations using your data for Q2.
4. 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 answers. Please include data to justify your answer.

Uploading to GradeScope: Upload your p2.cpp or cse.c file. If you added other files or made changes to other files, please submit those, too. If you modified the CMakeLists.txt, upload that as well. Include the report as a pdf along with your code. It will be ignored by the autograder script but made available to the instructor.

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 a maximum of 10 points total. Otherwise, assigned according to the rubric.

Rubric

ECE 566

10 points: Compiles properly with no warnings or errors
5 points: Code is well commented and written in a professional coding style
25 points: Fraction of benchmarks that are transformed properly (optimizations must be at least partially implemented to earn this). There are no points associated with the unit tests.
20 points: Most or all opportunities for optimization are exploited.
20 points: Fraction of secret tests that pass (these may overlap with provided tests)
20 points: Report with answers to the questions.

ECE 466

10 points: Compiles properly with no warnings or errors
5 points: Code is well commented and written in a professional coding style
25 points: Fraction of benchmarks that are transformed properly (optimizations must be at least partially implemented to earn this). There are no points associated with the unit tests.
20 points: Most or all opportunities for optimization are exploited.
20 points: Fraction of secret tests that pass (these may overlap with provided tests)
20 points: Report with answers to the questions.