EECS 583 - Homework 2

Assigned: Wednesday, September 27th, 2017
Due: Friday, October 13th, 2017

1 Summary

The goal of this homework is to extend the LLVM loop invariant code motion (LICM) optimization to identify more opportunities for optimization using *memory dependence speculation*. **The LLVM version you are required to work with is 3.3 release.**

1.1 Loop Invariant Code Motion (LICM)

The basic LICM pass in LLVM performs static analysis and hoist operations that can be moved outside the body of a loop without affecting the semantics of the program. For example, if we have the following code:

```
for (int i = 0; i < n; i++) {
  x = y + z;
  a[i] = 6 * i + x * x;
}
```

The calculation `x = y + z` and `x * x` can be moved outside the loop since within they are loop invariant – they do not change over the iterations of the loop. So the optimized code will be something like this:

```
x = y + z;
t1 = x * x;
for (int i = 0; i < n; i++) {
  a[i] = 6 * i + t1;
}
```

By doing so, you significantly reduce the number of operations you need to do inside the loop.

This is already implemented (partially) as one of the optimization passes in LLVM. Though not necessarily for the assignment, you can use the LLVM pass to see how LICM takes effect. Current LLVM licm pass hoists only one instruction (`x = y + z;`) in the above test example. The commands to use LLVM original LICM pass are as follows

```
clang -emit-llvm -o simple.bc -c simple.cpp
opt -basicaa -licm simple.bc -o simple.licm.bc
llvm-dis simple.bc
llvm-dis simple.licm.bc
```

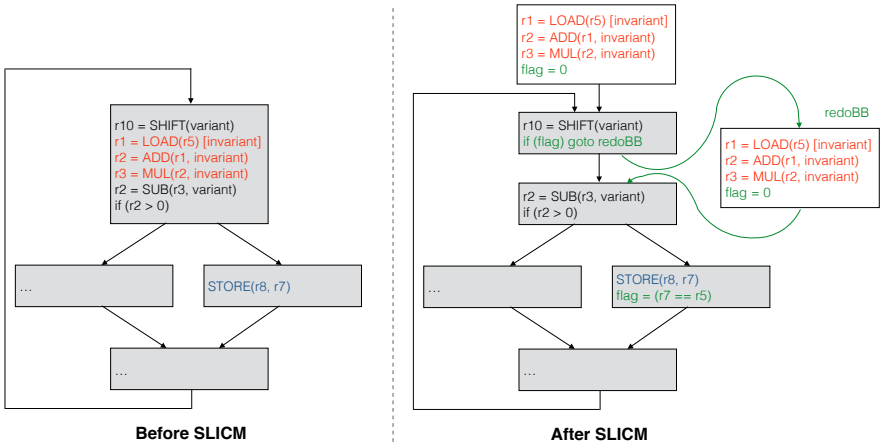

Figure 1: An example of speculative LICM. In the original code on left, the 3 red instructions cannot be hoisted because the `Load` instruction may depend on the `Store` instruction in blue, therefore cannot be hoisted. However, when we perform speculative LICM as shown on right, all these 3 red instructions can be hoisted outside the loop, with a special check on whether `r7` and `r5` point to the same memory location. If they do point to the same memory location, the corresponding fix up code in `redoBB` will be executed.

Look for differences in simple.ll and simple.licm.ll.

1.2 Speculative LICM

The basic LICM has certain limitation, because it is purely based on static analysis. During static analysis, the compiler often cannot tell whether `load` operations have memory dependencies or not. As a result, the basic LICM often cannot hoist memory operations, therefore misses the opportunity to hoist potential invariant operations. To overcome this limitation, your task in this homework in to perform speculative LICM by breaking unlikely memory dependencies. This will enable you to hoist not only more `load` operations out of the loops, but also other operations that solely depend on the hoisted `load` operations. Figure 1 shows an example of speculative LICM.

As you learned in homework 1, LAMP collects statistics on how often the addresses of two memory operations overlap for every pair of load/store instructions. You will use these statistics to identify profitable opportunities to perform speculative LICM. However, it is not that easy. The memory profile data can obviously be wrong and the situation where speculation is incorrect must be handled. Thus, you need to check for mis-speculation and correct execution when it happens. With LICM, mis-speculation handling can be accomplished by simply redoing a subset of the code that was hoisted. **Think carefully about this**.

2 Your Task

There are 2 parts in this homework, and you can use the code skeleton provided in the homework (i.e., `SLICMtemplate.cpp`) as a starting point for both parts.

In the first part (speculative LICM), you will be writing a compiler pass that hoist **all eligible** `load` operations (i.e., any `load` instruction which you can hoist and fix without changing the semantic of the program, except the loads that load the value of loop index variable, like `i`, into a register) in any loop, as well as their consumer operations that become invariant after hoisting the `load` operations. To maintain the correctness of the program, you need to insert the proper checks into the loop to identify mis-speculations and perform the necessary fix up. **This part will be graded.**

In the second part (intelligent speculative LICM), you will be designing an intelligent heuristic using LAMP profiling to only apply speculative LICM when profitable (i.e., there is a net gain in the performance by performing the transformation and handling mis-speculations) to achieve the best performance. **This part will not be graded, but we will have a contest on the performance improvement.**

2.1 Speculative LICM

In this part of the homework, you will be writing a compiler pass that hoist **all eligible** `load` operations (i.e., any `load` instruction which you can hoist and fix without changing the semantic of the program, except the loads that load the value of loop index variable, like `i`, into a register) in any loop, as well as their consumer operations that become invariant after hoisting the `load` operations. For example in the benchmark correct1.c, there is only one eligible load.

```
%7 = load i32* getelementptr inbounds ([100 x i32]* @a, i32 0, i64 97), align
4
```

To keep the program correct, you need to insert the proper checks into the loop to identify mis-speculations, and perform the necessary fix up.

This pass will be used to generate all the `*.slicm.bc` bitcode files required in submission.

There are many different ways to achieve speculative LICM, but here are a list of pointers that you may find useful depending on your implementation. Use of these functions are by no means required, and feel free to use any other code that exists in LLVM 3.3 or you develop.

In order to help you find out the eligible loads, find a sample code in the uploaded file - `findEligibleLoads.txt`. This code assumes that variable `I` is a load instruction, and ignores those load instructions that have a store writing into the same memory address as that of the load. This helps in ignoring the loads associated with index variables. It is your responsibility to figure out where to put this code in the `SLICMtemplate.cpp`.
Some useful common operations [1]
Basic LICM transformation [2]:

[1] http://llvm.org/releases/3.3/docs/ProgrammersManual.html#helpful-hints-for-common-operations
[2] `<LLVM_SRC_ROOT>/lib/Transforms/Scalar/LICM.cpp`

- `LICM::runOnLoop(...)` Processes current loop
  - Hoists/sinks invariants out of the loop body
  - Call `LICM::HoistRegion(...)` to handle hoisting for the loop
- `LICM::HoistRegion(...)` Processes loop's CFG
  - Makes sure an instruction has invariant operands and is safe to hoist, then calls `LICM::hoist(...)` to hoist an instruction
  - `LICM::HoistRegion(...)` called recursively until the whole loop's CFG has been processed
- `LICM::hoist(...)`
  - Hoists an instruction before the terminating instruction in the loop's pre-header basic block

To manipulate basic blocks [3]:

- `SplitBlock(...):` Split a basic block at a specified instruction, and return a pointer to the new basic block that starts with the specified instruction. The new basic block will be connected to the old basic block by an unconditional branch.
- `SplitEdge(...):` Insert a basic block between the 2 specified basic blocks.

To create and insert new instructions [4]:

- `LoadInst* LD = new LoadInst(flag, "load flag", BB1); :` Create a load instruction, and insert it at the end of the specified basic block.
- `Branch::Create(BB1, BB2, flag, BB3->getTerminator()); :` Create a branch instruction, and insert before BB3's terminating instruction that goes to BB1 if taken and BB2 if not taken
- `StoreInst* ST = new StoreInst(LD, var); ST->insertAfter(LD); :` Create a store instruction that stores the result of a load instruction to some variable.
- `ICmpInst* ICMP = new ICmpInst(BB1->getTerminator(), ICmpInst::ICMP_EQ, LD, ConstantInt::get(Int1Ty, 1));` Creates a comparison instruction that compares the value of destination register of LD to 1.

To create new variables:

- `AllocaInst* flag = new AllocaInst(Type::getInt1Ty(Entry->getContext(...), "flag", Entry->getTerminator()); :` Create a variable "flag" in the function entry block.

[3] `<LLVM_SRC_ROOT>/include/llvm/Transform/Utils/BasicBlockUtils.h`
[4] `<LLVM_SRC_ROOT>/include/llvm/IR/Instructions.h`

- `StoreInst* ST = new StoreInst(ConstantInt::getFalse(...), flag, Entry->getTerminator());` : Use the created variable "flag".

To copy an instruction, you can simply use the `clone()` function. However, the cloned instruction will use different registers compared to the original instruction, which may or may not be what you want. If you would like the cloned instruction to use the same set of registers, one way to achieve this is through the `mem2reg` pass in LLVM [5].

- Store the result of the hoisted instruction to a stack variable.
- Make sure the corresponding `AllocaInst` is in the function's entry basic block.
- After running your compiler pass, run the `-mem2reg` pass to convert these stack variables to registers.

2.2 Intelligent Speculative LICM

In this part of the homework, you will be designing an intelligent heuristic to determine which `load` instructions you want to hoist based on the LAMP profiling you did in homework 1. For example, one simple heuristic can be only hoisting when the dependence fraction of a `load` instruction is lower than certain threshold. You may also want to consider the number of instructions that solely depends on the `load` instruction, which can be moved out of the loop once you have hoisted the `load` instruction. This pass will be used to generate all the `*.intelligent-slicm.bc` bitcode files required in submission.

3 Testing and Grading

There are 2 sets of testing benchmarks: correctness and performance. The grading of this homework will **only** be based on the correctness benchmarks. However, we will have a contest among all the students on the performance benchmarks, that the person with the fastest average execution time across all performance benchmarks will be crowned the Optimization Champ and be awarded a prize. Note that correct execution results for all correctness and performance benchmarks (5 correctness + 4 performance) are required to qualify for the contest. However, incorrect results in performance benchmarks **do not** affect your grade.
Correctness testing benchmarks (`583hw2_test_correctness.tgz`). Please read and understand the code in the benchmarks:

- `correct1.c`: no runtime conflicts, a[97] should be speculatively hoisted.
- `correct2.c`: no runtime conflicts, a[97], multiply by 3 and add by 2 should be speculatively hoisted.

[5] http://llvm.org/releases/3.3/docs/Passes.html#mem2reg-promote-memory-to-register

- `correct3.c`: similar to `correct1.c` but has runtime conflicts.
- `correct4.c`: similar to `correct2.c` but has runtime conflicts.
- `correct5.c`: multiple independent chains to hoist and has runtime conflicts.

Performance testing benchmarks (`583hw2_test_performance.tgz`):

- `perf[1-3].c`: made up examples, a bit more complex than correct 5.
- `583wc`: benchmark from homework 1, should se some good speedups here.

4 Debugging

When working on your SLICM pass, you are going to want to verify that your pass is doing what you expect. Simply running the correctness benchmarks and seeing if it outputs the correct results, or relying on `errs()` debugging printouts while SLICM runs on some input code, are not going to be sufficient and probably will not reveal what might be going wrong in your code. I highly recommend you generate a visual output using `opt`'s ability to generate dotty files. To generate such files, the following will work:

```
opt -dot-cfg <filename>.bc >& /dev/null
```

This will generate a `.dot` file. Open the `.dot` file with `xdot`. `xdot` is available on the course servers. Just make sure you enable X11 forwarding when connecting: `ssh -X uniqname@eecs583{a,b}.eecs.umich.edu`. If you are working on MacOS, `graphviz` can achieve the same thing.

5 Submission

To submit your homework, send a single `.tgz` (gzipped tar file) into the directory `/hw2_submissions/` on `eecs583a.eecs.umich.edu` via `scp` (later version will automatically rewrite the earlier versions if you submit multiple times):

```
$ tar -cvzf ${uniquename}_hw2.tgz ${uniquename}_hw2
$ scp ${uniquename}_hw2.tgz ${uniquename}@eecs583a.eecs.umich.edu:/hw2_submissions/
```

Please name your tar file `${uniquename}_hw2.tgz`, and organize the directory inside in the following structure:

```
${uniquename}_hw2/
├── README
├── src/
│   ├── entire_directory_of_your_slicm_pass/
│   └── entire_directory_of_your_intelligent_slicm_pass/*
└── results/
    └── correct1.slicm.bc
```

```
├── correct1.intelligent-slicm.bc*
├── ...
├── correct5.slicm.bc
├── correct5.intelligent-slicm.bc*
├── perf1.slicm.bc*
├── perf1.intelligent-slicm.bc*
├── ...
├── perf3.slicm.bc*
├── perf3.intelligent-slicm.bc*
├── 583wc.slicm.bc*
└── 583wc.intelligent-slicm.bc*
```

Anything with * in the end is optional, if you do not want to participate in the performance contest. However, all these files are required if you would like to be qualified for the contest.

In `README`, describe what you did and what does/does not work. In `src/`, put the entire directories of your compiler passes (speculative LICM and intelligent speculative LICM if applicable) in there. For each directory, include all the Makefiles, autoconf-related files and everything required to build your pass. After submitting your homework, you should receive an email shortly (it may show up in spam).