EECS 583 – Homework 2

Fall 2018
Assigned: Wed, September 26, 2018
Due: Fri, October 12, 2018 (11:59:59 pm)

Frequent Path LICM

The goal of this homework is to extend the LLVM loop invariant code motion (LICM) optimization to identify more opportunities for optimization using *control flow profile information*. You will go beyond traditional LICM for instructions that are invariant along the most likely path through the loop even though they are not invariant when considering the entire loop body. Such instructions cannot be hoisted because traditional LICM must ensure that execution is correct regardless of the path taken through the loop body. However, by considering just a single path, one can be more aggressive and hoist additional instructions.

This optimization is a form of compile time speculation in that you are guessing the frequent path is followed. Whenever another path is taken, you must handle the mis-speculation and ensure correct execution. With LICM, mis-speculation handling can be accomplished by simply redoing the code that was speculatively hoisted.

Implementation

Extend LLVM's LICM (see llvm-source-dir/… /LICM.cpp) to perform speculative hoisting of "almost invariant" instructions. Almost invariant is defined as having a *single* source operand that is invariant along the most likely path through the loop but variant along 1 or more other, infrequent paths (the other source operand is completely invariant). As a simplifying assumption, you do not need to worry about pointers in this optimization. Rather, almost invariant instructions will be detected through explicit modification of a source operand from an infrequent block but invariant in the frequent blocks of the loop. Whenever the compiler speculates, it must have a repair mechanism to handle mis-speculations. With LICM, the hoisted instruction can simply be re-executed whenever an infrequent path is taken. Your implementation should consist of 4 parts:

1. Identify most likely path through a loop body. We will focus on innermost loops only. This can be accomplished by starting at the loop header and repeatedly following the most likely branch until a likely loop backedge is taken. All blocks along this path are classified as frequent and the rest in the loop body as infrequent.
2. Identify almost invariant instructions among the frequent blocks.
3. Create a heuristic to decide if hoisting the almost invariant instruction is profitable or not, and apply hoisting to profitable opportunities. Profit can be estimated by counting the number of the estimated dynamic instructions with and without hoisting.
4. Create and insert the repair code in case of mis-speculation.

Bonus Implementation

After performing the first hoist, e.g., a load, a number of dependent instructions may often become invariant and can also be speculatively hoisted. As a bonus, extend your baseline LICM implementation to hoist these dependent instructions and ensure that your repair code also properly handles these additional hoisted instructions. Again, your profitability heuristic should throttle this transformation to only apply when it is profitable. The bonus implementation is not required to complete and get full score on this homework, but those who successfully get this working will receive up to an additional 20% bonus score and bragging rights for a job well done.

Example

The following example demonstrates the basic and bonus versions of frequent path LICM. The leftmost column is the original code that contains a for loop with an infrequently taken if statement. Variable 'j' is only modified on an infrequent path, thus the load of j is a good target for frequent path LICM. The middle column shows the code after applying frequent path LICM to the load of j including the repair code to fix up loop execution when the infrequent path is taken. Note the repair code only needs to be executed on infrequent paths where j is modified. Finally, removing the load of j enables other dependent instructions to be hoisted including the load of A[j] and the multiplication by 23. This optimization is the optional part

of this homework referred to as the Bonus Implementation. The final result in shown in the rightmost column of the table below including the updated repair code.

int A[100], B[100], i, j = 99;	int A[100], B[100], i, j = 99;	int A[100], B[100], i, j = 99;
	int temp = j; /* hoisted load */	int temp = A[j] * 23; /* hoisted load and uses */
for (i=0; i<100; i++) {	for (i=0; i<100; i++) {	for (i=0; i<100; i++) {
/* Frequent path */ B[i] = A[j] * 23 + i;	/* Frequent path */ B[i] = A[temp] * 23 + i;	/* Frequent path */ B[i] = temp + i;
if (i%32 == 0)	if (i%32 == 0) {	if (i%32 == 0) {
j = i;	j = i;	j = i;
}	temp = j; /* repair code */	temp = A[j] * 23; /* repair code */
	}	}
	}	}

Contest

For the purposes of fun only, the person with the fastest average execution time across the benchmarks will be crowned EECS 583 F18 Optimization Champ. Note: correct execution results are required to qualify for the contest.

Submission

You should submit a single .tgz (gzipped tar) file into the directory /hw2_submissions on eecs583a.eecs.umich.edu via scp. Please name your tar file uniquename_hw2.tgz. Your tar file should contain:

1. Source code for your LLVM optimization pass. Please include all your code in the file HW2PASS.cpp. This should include your implementation of the baseline optimization and optionally your bonus implementation if you attempted this portion.
2. For each benchmark (both performance and correctness) provide bitcode files after performing your frequent path LICM optimization. Provide the bitcode files with just the baseline optimization (e.g., hw2correct1_base.bc) and optionally the bonus optimization (e.g., hw2correct1_bonus.bc) if you attempted the bonus portion of the assignment.
3. README that summarizes the status of your implementation, ie what works.

Use the following directory organization for your submitted tar file:

```
uniquename_hw2/
        HW2PASS.cpp (assuming you use Ze's template)
        benchmarks/
                hw2correct1_base.bc
                …
                hw2perf1_base.bc
                …
                hw2correct1_bonus.bc
                …
                hw2perf1_bonus.bc
                …
        README
```