CSE 583 – Homework 2

Fall 2025
Assigned: Wed, September 17, 2025
Due: Wed, October 08, 2025 (11:59:59 pm)

Late Submission Policy

Submissions are accepted a maximum of **two days** after the above specified deadline. For each day late, a **10% penalty** will be deducted from your score.

Frequent Path LICM

The goal of this homework is to extend the 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 frequent path LICM, mis-speculation handling can be accomplished by simply redoing the code that was speculatively hoisted.

Implementation (Correctness)

Implement an FPLICM pass to perform speculative hoisting of "almost invariant" instructions from the loop body. An instruction is said to be "almost invariant" when its source operands are invariant along the frequent (most likely) path of the loop, but may be variant along other infrequent paths. To simplify your code, you do not need to worry about pointers (i.e. you can assume there is no pointer aliasing), you just need to worry about explicit modification of source operands along infrequent paths. Since this is a compiler speculation, we need to add a repair mechanism to handle mis-speculations. For LICM, the repair mechanism involves repeating the hoisted instructions along the infrequent path. Your pass should perform three steps:

1. Identify the loops, and for each loop identify the "most likely" path through the loop body. This is done by starting at the loop header and repeatedly choosing the >=80% branch until the backedge is taken. **Such a frequent path exists in all of our test cases**. Anything not on the frequent path is said to be infrequent.
2. Identify load instructions on the frequent path that are "*almost invariant*" and hoist them. (For the correctness part, you only need to hoist "almost invariant" **load instructions**)
3. Create and insert the repair code in the infrequent paths to handle mis-speculation. At the end of this step, the output of the original code and the optimized code should be identical.

Bonus Implementation (Performance)

After hoisting the load instructions, a number of dependent instructions may become invariant and can also be speculatively hoisted. For bonus points, extend your baseline implementation to hoist these dependent instructions whilst ensuring program correctness. You can further create a heuristic to decide if hoisting is profitable and do so only when it is. Profit can be estimated by the reduction in the dynamic instruction count after applying your pass. The bonus implementation is not required to score 100% on this homework, but those who successfully get it working will receive up to an additional 20% on the homework score and bragging rights for a job well done.

Example

The following example demonstrates the basic and bonus versions of frequent path LICM at the source code-level. 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, after maximal hoisting is shown in the rightmost column of the table below including the updated repair code.

Original Code	First load hoisted	First load and uses hoisted
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 = A[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] = 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 = A[j]; /* repair code */	temp = A[j] * 23; /* repair code */
	}	}
	}	}

Contest

As an added motivation, the person with the fastest average execution time across the benchmarks will be crowned CSE 583 Fall 2025 Optimization Champ and be awarded a prize. Note: To qualify for the contest, all execution results are required to be correct. It does not matter how fast you are if you have broken the code!

Submission

You should submit only the completed **hw2pass.cpp** file on Gradescope. Once you submit your code, it will be run on the six correctness benchmarks and the four performance benchmarks discussed above. For the correctness benchmarks, you will see a score based on whether the original program's behavior was retained and whether the loads were hoisted correctly. For the performance benchmark, you will see a score of 0 on the autograder along with the measured speedup because of your pass. The performance part will be graded manually after the submission window. Keep the following in mind:

1. Verify that your code works locally or on the server before uploading to Gradescope. Because of the way the testing is happening under the hood, you may not see all the errors that occur if your code, say, fails to compile, which can make debugging only using autograder output difficult.
2. Ensure that you do not leave dead instructions in your code after hoisting.
3. The score the autograder gives you may not be the final score and may be changed based on the severity of the mistakes made (which means that even if you get 0 on the autograder, you can get a high score if your errors are judged to be minor).