CS 380C: Advanced Topics in Compilers
Assignment 4: Loop invariant code motion with LLVM

Feb 25th, 2025

Due date: March 13th, 2025
Late submission policy: Submission can be at the most 2 days late. There will be a 10% penalty for each day after the due date (cumulative).

A compiler analyzes and optimizes programs in passes; each pass scans or traverses the intermediate code to analyze or transform it. Analysis passes are typically used to collect information about a program that helps in ensuring that an optimization pass is safe (not alter the meaning of the program) or determining whether an optimization pass could be beneficial. An optimization pass then transforms the code.

The goal of this assignment is to familiarize you with the LLVM compiler by implementing two LLVM passes to (i) find loop-invariant computations (analysis), and (ii) move them out of loops (optimization).

1 Pass 1: properties of loops

In this analysis pass, you must implement code to report the following information about all loops that are used in a program.

1. Function: name of the function containing this loop.
2. Loop depth: 0 if it is not nested in any loop; otherwise, it is 1 more than that of the loop in which it is nested in.
3. Contains nested loops: determine whether it has any loop nested within it.
4. Number of top-level basic blocks: count all basic blocks in it but not in any of its nested loops.
5. Number of instructions: count all instructions in it, including those in its nested loops.
6. Number of atomic operations: count atomic instructions in it, including those in its nested loops.
7. Number of top-level branch instructions: count branch instructions in it but not in any of its nested loops.

For each loop, print a line to standard error in the following format (case and space sensitive):

```
<ID>:  func=<str>, depth=<no>, subLoops=<str>, BBs=<no>, instrs=<no>, atomics=<no>, branches=<no>
```

`<ID>` represents a global counter for each loop encountered, starting at 0. `<str>` represents a string for function name or boolean value - “true” or “false”. `<no>` represents a number.

2 Pass 2: hoisting loop-invariant computations out of loops

In this pass, you need to optimize loops by hoisting loop-invariant code out of the loop.
Input: An LLVM loop L.

Output: The loop, with loop-invariant computations hoisted out of that loop and its inner loops as far as possible.

Preconditions: The loop simplification pass should have been performed on the function where the loop is present. This ensures that every loop has a preheader. It does not ensure that a "while" loop is converted into a do-while loop nested inside an IF; you should not assume that. You will also need information about natural loops and about dominators. (Another pass that can make LICM more effective is reaassociation, but that is not requred for the algorithm and is not included here; you should run it yourself before your pass). This is done through a combination of adding the pass during the registration of your pass, and the `AnalysisManager::getResult` function in the pass.

The starter code provided already adds the required transform passes. If you would like to take a look or add other passes, it is in the `getLooperPluginInfo` function towards the bottom of `looper.cpp`.
NOTE: You will still need to add the analysis passes that you want run.

Algorithm: The goal of this algorithm is to hoist as many loop-invariant computations out of loops as possible. We focus only on register-to-register LLVM computations, i.e., those that do not read or write memory. We do not try to move out computations that have no uses within the loop: these could be moved after the loop (at all loop exits), but that requires patching up SSA form. We say it is safe to hoist a computation out of a loop only if it is executed at least once in the original loop (when the loop is entered); this condition is checked using dominator information. The full list of criteria are given below.

```
LICM(L)
{
  // Each Loop object also gives you a preheader block for the loop.
  for (each basic block BB dominated by loop header,
                        in preorder on dominator tree) {
    if (BB is immediately within L) { // not in an inner loop or outside L
      for (each instruction I in BB) {
        if (isLoopInvariant(I) && safeToHoist(I))
          move I to pre-header basic block;
      }
    }
  }
}
```

`isLoopInvariant(I):`
An instruction is loop-invariant if both of the following are true:

1. It is one of the following LLVM instructions or instruction classes:

   `binary operator, shift, select, cast, getelementptr.`

All other LLVM instructions are treated as not loop-invariant. In particular, you are not moving these instructions - `terminators, phi, load, store, call, invoke, malloc, free, alloca, vanext, vaarg`.

2. Every operand of the instruction is either (a) constant or (b) computed outside the loop.

`safeToHoist(I)`:
An instruction is safe to hoist if either of the following is true:

1. It has no side effects (exceptions/traps). You can use `isSafeToSpeculativelyExecute()` (you can find it in `llvm/Analysis/ValueTracking.h`).
2. The basic block containing the instruction dominates all exit blocks for the loop. The exit blocks are the targets of exits from the loop, i.e., they are outside the loop.

Comments on the safety conditions:
Unlike the conditions discussed in class (and in lecture notes):

- You are using relaxed conditions for a non-excepting expression: hoisting it out of a loop (even a while loop) is fine even if it does not dominate all exits, i.e., it may lengthen some path on which it was never executed before.
- You are hoisting out an excepting expression (only) if it dominates all exit blocks. This guarantees that you will not cause a trap unless it would have been caused by the original program. However, you may potentially reorder two trapping instructions if you hoist one of them out of the loop and not the other. This means that the algorithm is safe for C, Fortran, C++, but not for Java or C#.

3 Getting Started with LLVM

Familiarize yourself with the LLVM compiler infrastructure since you will be using it for the coming assignments too. You can find all the necessary documentation here:

http://llvm.org/docs/

You can use the following LLVM manuals to learn more about the compiler:

LLVM command line tools	http://llvm.org/docs/CommandGuide/
LLVM IR Reference Manual	http://llvm.org/docs/LangRef.html
LLVM Programmer's Manual	http://llvm.org/docs/ProgrammersManual.html
Writing an LLVM Pass	https://llvm.org/docs/WritingAnLLVMNewPMPass.html
LLVM testing infrastructure	http://llvm.org/docs/TestingGuide.html
Downloading and building LLVM	http://llvm.org/docs/GettingStarted.html

Read these manuals very selectively, or you will spend far too much time on them!

Download the precompiled LLVM-19 binaries from the github release artifacts found here or your favorite package manager (instructions for apt found here).

You can use any machine for development. You will need roughly 2 GB of disk space on your machine for the prebuilt LLVM binaries. To save you having to compile the whole of LLVM you will be writing your pass as a "plugin pass". If you want more information, you can find it in the documentation provided above.

4 Implementation Guidelines

Please follow these guidelines precisely because the grading scripts will be based on it.

1. Download the starter code from here (https://github.com/kayvan61/CS380C-Assignment4).
2. Replace any instances of `UTEID` in the provided starter code with your EID. (Filenames and CMakeLists.txt too)
3. Run CMake using `cmake -S . -B ./build -DCMAKE_BUILD_TYPE=Debug`
4. Run `make -C ./build` to build the pass shared objects.
5. Run your analysis pass using this command:

   ```
   opt -load-pass-plugin libloop-analysis-pass.so -passes="UTEID-loop-analysis-pass" <$INPUT >/dev/null
   2>$OUTPUT
   ```

   $INPUT will be a LLVM IR or LLVM bitcode file (`.ll or .bc`).
6. Run your optimization pass using this command:

   ```
   opt -load-pass-plugin ./build/libloop-opt-pass.so -passes="UTEID-loop-opt-pass" <$INPUT >/dev/null
   2>$OUTPUT
   ```

   $INPUT will be a LLVM IR or LLVM bitcode file (`.ll or .bc`).

Try using the C++ Standard Template Library, if you haven't already - it can enable you to use effective data structures quickly (e.g., sets, maps, lists, and vectors). LLVM also offers many data structures in its library. See the LLVM Programmer's Manual for choosing an appropriate data structure. If you find you are writing code for doing something basic within LLVM, it is quite likely that the code for this exists already - use it. However you are not allowed to copy source code from anywhere, including other LLVM source files. You can include the header files and call those functions if you think it does precisely what you want. If you are not sure whether you can use something or if you think some LLVM source code is making the assignment trivial, then please let us know on Piazza (you can use a private note).

IMPORTANT: DO NOT READ `$SRC_ROOT/lib/Transforms/Scalar/LICM.cpp`

5 Testing

You are responsible for writing test cases to test your passes. Write test cases such that you can test various aspects of your passes; start off with simple ones and then add more complex ones. Feel free to use LLVM test programs. We can use any program for testing your pass.Your program must be robust (no crashes or undefined behavior) and precise (correct output). Please test your code thoroughly.
You may find it useful to view what LLVM generates for some example source code. Try out godbolt.org or compile with `clang++ -S -emit-llvm <cpp file>`. This will give human readable LLVM-IR (`.ll` file). Additionally, most every llvm object (that it makes sense for) has a `dump()` method for easy debugging.

6 Deliverables

Submit (to canvas) an archive (preferably, `.tar.gz`) of the source directory containing your source code, build script (`CMakeLists.txt`) and `README` file. Please do not submit the built binary.
The `README` file should mention all the materials that you have read or used for this assignment, including LLVM documentation and source files; you can only skip mentioning header files that you have explicitly included in your source code. Mention the status of your submission, in case some part of it is incomplete. You can also include feedback, like what was challenging and what was trivial. Please also mention your project partner.
You can also submit the test cases for which your pass worked (in the same directory); if they are publicly available, then just mention them in the README file.
Please feel free to help each other with the build system since that is not focus of this class.

7 Grading

We will do the following to grade your assignment:

1. Unpack the archive you submit into a new folder, which will contain a `CMakeLists.txt`.
2. Run `cmake` and `make` inside your sub-directory in the new folder.
3. Use `opt` as mentioned in the implementation guidelines to test your pass on a few input programs. Your program should not crash.
4. Compare (`diff`) the output (to standard error) against the expected output. It should match.

Note that this will be done through scripts. So, please follow the output and implementation guidelines precisely.

Any clarifications and corrections to this assignment will be posted on Piazza.