USCC
Programming Assignment 5 – Optimization Passes

By: Sanjay Madhav
University of Southern California

Useful Links

In addition to the standard LLVM documentation, for this programming assignment it is also useful to consult “Writing an LLVM Pass”.

Introduction

In the LLVM framework, optimizations are implemented as a series of passes which transform the LLVM bitcode. Each optimization pass can have set dependencies, which ensures that passes are executed in the intended order. There are several different types of optimization passes, but in this programming assignment, you will implement two function passes (constant branch folding and dead block removal) and one loop pass (loop invariant code motion).

A *function pass* is a local optimization pass that is executed once per each function in the code module. Since it is limited to a single function, it will not be able to optimize interactions between several different functions. Rather, it can only iterate through the basic blocks of a single function and apply whichever optimizations are desired.

A *loop pass* is a local optimization pass that is executed once per loop inside of a function. This is handy because it will automatically identify back edges and loops for you. While it certainly would be possible to manually determine this in a function pass, it’s much simpler to apply loop-focused optimizations in a loop pass.

In USCC, optimization passes can be invoked with the `-O` (that is, an uppercase letter O) command line switch. For example, assuming you are in the tests directory, the following would compile `opt01.usc`, run the optimization passes, and print the resulting bitcode to stdout:

```
$ ../bin/uscc -p -O opt01.usc
```

The provided starting code has one function pass already implemented – *constant propagation* – which is implemented in `opt/ConstantOps.cpp`. Constant propagation is a peephole optimization that looks for certain instructions, such as arithmetic, that are operating on constant values. These instructions are then replaced with the resultant constant value. For example, an add instruction between the constants 5 and 10 would have its uses replaced with the constant value of 15.

In practice, a great deal of constant propagation in LLVM occurs when you construct the bitcode – the factory methods used to create the instructions check for constant values and propagates them automatically. However, it is still possible in some cases to end up with unpropagated constants, which is what this optimization pass aims to eliminate.

You should study the code in `ConstantOps.cpp` before continuing. Notice how there are two functions implemented – this is the minimum required for any pass. The `getAnalysisUsage` function is used to inform the LLVM Pass Manager how a particular optimization pass behaves.

For `ConstantOps`, the only behavior noted is that it preserves the CFG. This is because the pass will not alter any of the edges between blocks. The `getAnalysisUsage` function can also be used to specify dependencies – in this case, `ConstantOps` does not depend on any other passes.

The other function in `ConstantOps.cpp` is `runOnFunction`. This represents the actual optimization pass code, and is called once per function. Since this is called once per function, you are not allowed to have any data persist between multiple function calls (such as any static data). All data must be local to a single invocation.

The overall structure of `ConstantOps::runOnFunction` is similar to many function passes – it iterates through every basic block in the function, and then every instruction in the basic block. It uses `isa` or `dyn_cast` to identify instructions of interest, and then uses the `replaceAllUsesWith` member function to replace these instructions. One thing that's very important to note is that `eraseFromParent` is not called during the initial iteration, because this will invalidate the iterator. Instead, instructions to be erased are added to a set, and then erased once all instructions have been visited. Finally, `runOnFunction` should return `true` if the function was changed in any way.

Once you've studied `ConstantOps.cpp`, it's time to implement your own optimization passes. Note that all of these optimization passes already exist in one form or another in the base LLVM code. However, do not copy the implementation from the base LLVM code because it will be obvious, defeats the purpose of the exercise, and lacks academic integrity.

In addition to checking your output vs. the expected output, it is recommended you run the `testOpt.py` test suite after implementing each pass. This test suite is like `testEmit.py`, except it runs with the `-O` flag, which means your optimization passes will execute. This will help catch any broken code in your pass.

To run the test suite, you would execute the following from the tests directory:

```
$ python testOpt.py
```

Constant Branch Folding

From the tests directory, run the following command:

```
$ ../bin/uscc -p -O opt01.usc
```

The output should look something along the lines of:

```
@.str = private unnamed_addr constant [4 x i8] c"%d\0A\00"
declare i32 @printf(i8*, ...)
define i32 @main() {

entry:
  br i1 true, label %if.then, label %if.else
if.then:                                          ; preds = %entry
  br label %if.end
if.else:                                          ; preds = %entry
  br label %if.end
if.end:                                           ; preds = %if.else, %if.then
  %b = phi i32 [ 100, %if.else ], [ 90, %if.then ]
  %0 = call i32 (i8*, ...)* @printf(...)
  ret i32 0
}
```

The instruction highlighted in red is notable because it is a conditional branch which is conditional on a constant value. Since the condition is `true`, the left successor (`%if.then`) will always be selected. There is no way `%if.else` can be selected. However, because the branch as written is conditional, the `%if.end` block must have a phi node to determine the value of `%b`.

Constant branch folding is an optimization where conditional branches on constant values are converted into unconditional branches. In addition to simplifying the branches, in some cases this will eliminate phi nodes (though not in this particular instance). This pass should be implemented in `opt/ConstantBranch.cpp`.

getAnalysisUsage

The `getAnalysisUsage` for this pass should be:

```
Info.addRequired<ConstantOps>();
```

This denotes that you want the `ConstantBranch` pass to only execute once the `ConstantOps` pass has been executed on the function. This ensures that all constants have been propagated prior to inspecting branch instructions.

You *do not* want to specify `setPreservesCFG` for this pass, because it will potentially alter edges in the CFG.

Pseudocode for runOnFunction

```
foreach BasicBlock BB in F...
   foreach Instruction i in BB...
      if i isa BranchInst
         cast i to BranchInst br
         if br is conditional and its condition isa ConstantInt...
            Add br to removeSet
```

```
foreach BranchInst br in removeSet...
   if br's condition is true...
      Create a new BranchInst that's an unconditional branch to the left successor
      Notify the right successor that br's parent is no longer a predecessor
   else...
      Create a new BranchInst that's an unconditional branch to the right successor
      Notify the right successor that br's parent is no longer a predecessor
   Erase br
```

Tips

- You can use either `isa` plus `cast` *or* `dyn_cast` to see if it's a `BranchInst*`
- Instruction has a `getParent` member function to get the containing `BasicBlock*`
- `BranchInst` has the following useful member functions:
  - `isConditional` – Returns whether or not the branch is conditional
  - `getCondition` – Returns the `llvm::Value*` of the condition
  - `getSuccessor(int)` – Returns the successor `BasicBlock*` (0 for left, 1 for right)
- `BasicBlock` has a `removePredecessor` member function

Expected Output

Once the `ConstantBranch` pass is implemented, the following command:

```
$ ../bin/uscc -p -O opt01.usc
```

Should roughly yield:

```
@.str = private unnamed_addr constant [4 x i8] c"%d\0A\00"
declare i32 @printf(i8*, ...)

define i32 @main() {
entry:
  br label %if.then
if.then:                                          ; preds = %entry
  br label %if.end
if.else:                                          ; No predecessors!
  br label %if.end
if.end:                                           ; preds = %if.else, %if.then
  %b = phi i32 [ 100, %if.else ], [ 90, %if.then ]
  %0 = call i32 (i8*, ...)* @printf(...)
  ret i32 0
}
```

The entry block now has an unconditional branch instead of a conditional one. However, the phi node is still there, and `%if.else` is an unreachable basic block. This will be fixed in the next optimization pass you will implement.

Dead Block Removal

The purpose of this pass is to remove any basic blocks that are unreachable. The way you will determine if a basic block is unreachable is to first perform a DFS from the entry block. Once all blocks have been visited in the DFS, you will have a set of all of the reachable blocks. You will then iterate through all the blocks inside the function, and remove any which are not in the reachable set.

This pass should be implemented in `opt/DeadBlocks.cpp`.

getAnalysisUsage

You want the `DeadBlocks` pass to execute after `ConstantBranch`. Thus, add the following:

```
Info.addRequired<ConstantBranch>();
```

Pseudocode for runOnFunction

```
Perform a DFS from the entry block, adding each visited block to the visitedSet
foreach BasicBlock BB in F...
   if BB is not in the visitedSet
      Add BB to unreachableSet
foreach BasicBlock BB in unreachableSet...
   foreach successor to BB...
      Tell the successor to remove BB as a predecessor
   Erase BB
```

Tips

- You can use `df_ext_begin` and `df_ext_end` to use the DepthFirstIterator, which automatically adds visited blocks to a `set` of your choosing (i.e., `visitedSet` in the pseudocode above)
- To iterate over the successors of a `BasicBlock`, use `succ_begin` and `succ_end`

Expected Output

Once the `DeadBlocks` pass is implemented, the following command:

```
$ ../bin/uscc -p -O opt01.usc
```

Should roughly yield:

```
@.str = private unnamed_addr constant [4 x i8] c"%d\0A\00"
declare i32 @printf(i8*, ...)

define i32 @main() {
entry:
  br label %if.then
if.then:                                          ; preds = %entry
  br label %if.end
if.end:                                           ; preds = %if.then
  %0 = call i32 (i8*, ...)* @printf(...)
  ret i32 0
}
```

Of course, you could further optimize this such that all three basic blocks in main are combined into one, but don't worry about that particular optimization.

Loop Invariant Code Motion

For the final optimization pass, you will implement a basic form of *loop invariant code motion* (LICM) in `opt/LICM.cpp`. Recall that the purpose of LICM is to find computations performed inside of the loop that do not depend on the loop in any way. These computations can then be hoisted out of the loop into the loop's preheader block.

LICM relies on certain information such as dominator trees. Because the algorithm for constructing dominator trees is relatively complex, you will utilize LLVM's built-in pass to generate the dominator tree for you. This simplifies the work that needs to be done rather significantly.

In LLVM, loop optimization passes still have a `getAnalysisUsage` function. However, instead of `runOnFunction`, they have the appropriately-named `runOnLoop` function.

Before you implement this pass, take a look at the code in `opt05.usc`:

```
int main()
{
   int i = 10;
   char str[] = "HELLO WORLD!";
   char letter = str[1];

   while (i > 0)
   {
      char result = letter + 32;
      printf("%c\n", result);
      --i;
   }

   return 0;
}
```

Notice how `result` is calculated inside of the loop, but it depends on `letter` which is assigned prior to the loop. This is a prime candidate for LICM, and can be confirmed by inspecting the bitcode.

Run the following command in the tests directory:

```
$ ../bin/uscc -p -O opt05.usc
```

There is a lot of bitcode, but you are only interested in the `%while.body` block, which should look like:

```
while.body:                                          ; preds = %while.cond
  %conv = sext i8 %2 to i32
  %add = add i32 %conv, 32
  %conv1 = trunc i32 %add to i8
  %conv2 = sext i8 %conv1 to i32
  %4 = call i32 (i8*, ...)* @printf(...)
  %dec = sub i32 %i, 1
  br label %while.cond
```

In this case, `%2` corresponds to the `letter` variable in `opt05.usc`, and thus is declared in the entry block. The subsequent instructions, `%add`, `%conv1`, and `%conv2` ultimately depend on `%2`, as well. This means all of four of these instructions (in red) can be hoisted out of the loop.

getAnalysisUsage

The `getAnalysisUsage` for this pass is more complex because there are dependencies on built-in passes:

```
// LICM does not modify the CFG
Info.setPreservesCFG();
// Execute after dead blocks have been removed
Info.addRequired<DeadBlocks>();
// Use the built-in Dominator tree and loop info passes
Info.addRequired<DominatorTreeWrapperPass>();
Info.addRequired<LoopInfo>();
```

Implementation of runOnLoop

First, if you open `opt/Passes.h`, you will see that there are some member variables declared for the LICM class. This is because the LICM implementation will utilize a preorder traversal, which means recursion, which means the implementation can't be entirely in `runOnLoop`.

At the start of `runOnLoop`, you need to initialize the member variables as follows:

```
mChanged = false;
// Save the current loop
mCurrLoop = L;
// Grab the loop info
mLoopInfo = &getAnalysis<LoopInfo>();
// Grab the dominator tree
mDomTree = &getAnalysis<DominatorTreeWrapperPass>().getDomTree();
```

The `getAnalysis` method is how you get the necessary information (such as the dominator tree) from built-in passes.

Once the member variables are initialized, you then need to implement a few more functions.

isSafeToHoistInstr

Create a member function called `isSafeToHoistInstr` that takes in an `llvm::Instruction*` as a parameter, and returns a `bool` on whether or not the instruction is safe to hoist. An instruction as safe to hoist if the following conditions are true:

1. It has loop invariant operands. Use `mCurrLoop->hasLoopInvariantOperands` to determine this.
2. It is safe to "speculatively execute" – which is defined as an instruction that does not have any unknown side effects. You can use the global `isSafeToSpeculativelyExecute` function for this.
3. It is one of the following instruction classes: `BinaryOperator`, `CastInst`, `SelectInst`, `GetElementPtrInst`, and `CmpInst`. This list of instructions was taken from the LLVM implementation of LICM.

hoistInstr

Next, create a member function called `hoistInstr` that takes in an `llvm::Instruction*` as a parameter, and hoists the instruction to the preheader block. To get this block, you can use `mCurrLoop->getLoopPreheader()`. Once you have the preheader block, you can use `getTerminator()` to get the terminator of that block. Then use the `moveBefore` member function to move the `llvm::Instruction*` (that came in as the parameter to `hoistInstr`) to be prior to the terminator.

Finally, `hoistInstr` should also set the `mChanged` member variable to `true`, since you've now changed the instructions in the loop.

hoistPreOrder

Now create a member function called `hoistPreOrder`, that takes an `llvm::DomTreeNode*` as a parameter.

The pseudocode for this function is as follows:

```
Get the BasicBlock BB associated with DomTreeNode...

if BB is in the current loop...
   foreach Instruction i in BB...
   Save i in currentInstr
   i++
   if isSafeToHoist(currentInstr)
      hoistInstr(currentInstr)

foreach child of DomTreeNode...
   hoistPreOrder(child)
```

It's extremely important to note that the iteration differs from how you've done it previously. You want to save the current instruction, move to the next instruction, and then see if you can hoist the current one. The reason this is necessary is because if you hoist the instruction and then increment `i`, the iterator will actually point inside the preheader block, and your code will get stuck in an infinite loop. By incrementing past the current instruction, you avoid this issue.

Tips for hoistPreOrder

- Use `getBlock` on the node to get the `BasicBlock*` associated with it
- To figure out if the block is in the current loop, you can use `mLoopInfo->getLoopFor()` and see if that loop corresponds to `mCurrLoop`. This step is important to make sure other loops aren't constantly revisted.
- You can get the children of a dominator tree node using the `getChildren` member function, which returns a vector of DomTreeNode*

Calling hoistPreOrder

Once all of these functions are implemented, `runOnLoop` should be updated so after all the member variables are set, you should call `hoistPreOrder` passing in the `DomTreeNode*` associated with the loop header. You can get the loop header with `mCurrLoop->getHeader()` and then from that use `mDomTree->getNode` to get the `DomTreeNode*` corresponding to that basic block.

Expected Output

Once LICM is implemented, the following command:

```
$ ../bin/uscc -p -O opt05.usc
```

Should yield an entry block that looks something like:

```
entry:
  %str = alloca [13 x i8], align 8
  %0 = getelementptr inbounds [13 x i8]* %str, i32 0, i32 0
  call void @llvm.memcpy.p0i8.p0i8.i64(...)
  %1 = getelementptr inbounds i8* %0, i32 1
  %2 = load i8* %1
  %conv = sext i8 %2 to i32
  %add = add i32 %conv, 32
  %conv1 = trunc i32 %add to i8
  %conv2 = sext i8 %conv1 to i32
  br label %while.cond
```

And the corresponding `%conv`, `%add`, %conv1, and %conv2 instructions should no longer be part of %while.body.

Conclusion

Make sure that once you have implemented all of the optimization passes, you run the testOpt.py test suite and ensure that nothing has broken. If everything works, congratulations! You have finished the fifth (and final) USCC programming assignment.