Table of Contents

[Algorithms Implemented 2](#_Toc165236581)

[Loop Analysis 2](#_Toc165236582)

[Loop Invariant Code Motion (LICM) 2](#_Toc165236583)

[Sparse Conditional Constant Propagation (SCCP) 2](#_Toc165236584)

[Code Status 3](#_Toc165236585)

[Pass Statistics 4](#_Toc165236586)

[Experimental Results 5](#_Toc165236587)

[Conclusion 6](#_Toc165236588)

[References 7](#_Toc165236589)

# Algorithms Implemented

## Loop Analysis

Following the algorithm explained at <https://pages.cs.wisc.edu/~fischer/cs701.f14/finding.loops.html>

Setup:

I iterate through each BasicBlock within the function we are analyzing and give each a name. We then create an empty standard c++ map container to hold our natural loops. Our key is the label of the loop-header and the value is a set of all nodes within the loop body. I am also utilizing the LLVM DominatorTree class provided by the LLVM FunctionAnalysisManager.

Analysis:

I iterate through each BasicBlock within a function; for each BasicBlock I check all of its successors. If a BasicBlock’s successor dominates it, we know we have found a back edge. To verify the BasicBlock (Block A, herein) to which the backedge leads creates a natural loop, I check if Block A dominates all BasicBlock’s between it and the block which had a back edge to it (Block B, herein). If it does, then I know that the loop is a natural loop and that Block A is the header to said loop.

Using this information, create a std::set to contain all of the BasicBlock’s within the loop body and first add Block A to the set. From there I utilize a while loop and stack to work my way up from Block B and collect all the BasicBlocks between it and block A. I run the loop for as long as the stack is not empty; before the loop I add Block B to the stack. In the loop I pop the basic block and check if it has already been added to the loop body, if not, I insert it into the loop body and iterate through all its predecessors. I then add each of its predecessors onto the stack and repeat this process until the stack is empty which ensures that all nodes have been captured.

After the end of the while, I add the set of nodes (LoopBody) to the map I created with the key value being the llvm::StringRef of the name of the loop header Basic Block.

## Loop Invariant Code Motion (LICM)

Following algorithm explained at <https://www.cs.utexas.edu/~pingali/CS380C/2020/assignments/assignment5/index.html>

Setup:

I retrieve the dominator tree

Analysis:  
I iterate through the map data structure built in the loop analysis phase. The keys represent the name of the header block and the value is an ordered set of the BasicBlocks within the loop body (including the loop header). For each iteration over a loop, I create a preheader BasicBlock for the loop header. Then I begin iterating through each BasicBlock of the loop body and for each BasicBlock in the loop body I iterate through each of their instructions.

I check each instruction for eligibility for hoisting. I first check if an instruction has any side effects. If it does, I check that it dominates all exit blocks. If it does not it is not eligible, else I continue to check if the instruction is loop invariant (operands are constant, operands are defined before the loop, or operands refer to variables that are loop invariant). If this instruction is identified as loop invariant, it is eligible to be hoisted. I then hoist the instruction out of the loop by moving the instruction to the BasicBlock created as the preheader to the loop header.

## Sparse Conditional Constant Propagation (SCCP)

# Code Status

# Pass Statistics

# Experimental Results

# Conclusion

# References

TAs:

Wenqi

Loop Analysis:

<https://pages.cs.wisc.edu/~fischer/cs701.f14/finding.loops.html>

LICM:

<https://www.cs.utexas.edu/~pingali/CS380C/2020/assignments/assignment5/index.html>