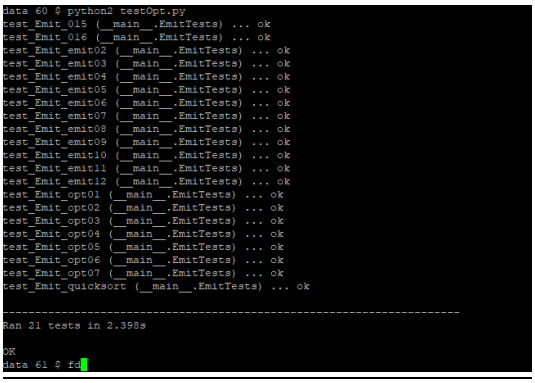
Report

won29@purdue.edu

Jong Yun Won

CS352 PA6

Test Results (Screenshot)



```
data 61 $ ../bin/uscc -p -O opt01.usc
; ModuleID = 'main'

@.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(i8* getelementptr inbounds ([4 x i8]* @.str, i32 0, i32 0), i32 90)
  ret i32 0
}
data 62 $ ../bin/uscc -p -O opt05.usc
; ModuleID = 'main'

@.str = private unnamed_addr constant [4 x i8] c"%c\0A\00"
@.str1 = private unnamed_addr constant [13 x i8] c"HELLO WORLD!\00"

declare i32 @printf(i8*, ...)

define i32 @main() {
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(i8* %0, i8* getelementptr inbounds ([13 x i8]* @.str1, i32 0, i32 0), i64 13, i32 1, i1 false)
  %1 = getelementptr inbounds i8* %0, i32 1
  %2 = load i8* %1
  %conv = sext i8 %2 to i32
  %add = add i32 %conv, 32
  %conv2 = trunc i32 %add to i8
  %conv3 = sext i8 %conv2 to i32
  br label %while.cond

while.cond:                                       ; preds = %while.body, %entry
  %Phi = phi i32 [ %dec, %while.body ], [ 10, %entry ]
  %gt = icmp sgt i32 %Phi, 0
  br i1 %gt, label %while.body, label %while.end

while.body:                                       ; preds = %while.cond
  %3 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @.str, i32 0, i32 0), i32 %conv3)
  %dec = sub i32 %Phi, 1
  br label %while.cond

while.end:                                        ; preds = %while.cond
  ret i32 0
}

; Function Attrs: nounwind
declare void @llvm.memcpy.p0i8.p0i8.i64(i8* nocapture, i8* nocapture readonly, i64, i32, i1) #0

attributes #0 = { nounwind }
data 63 $
```

ConstantBranch

First in getAnalysisUsage(), we included that the ConstantOps pass was required before this pass.

```
// PA6: Implement
Info.addRequired<ConstantOps>();
```

Now referring to the handout, we iterated through the basic blocks and for each instruction in the basic block, we checked if it was branch instruction or not

```
for (auto &bb : F) {
    for (auto &inst : bb) {
        if (isa<BranchInst>(inst)) {
```

Then, if so, we checked if the instruction was condition, and the condition was constant int, in which case, it was added to the removeSet.

```
auto br = dyn_cast_or_null<BranchInst>(&inst);
if (br->isConditional() && isa<ConstantInt>(br->getCondition())) {
    removeSet.insert(br);
}
```

Finally, we iterate through the removeSet and if the removeSet is not zero, set changed to True. For each removeSet instruction, we either created a branch to the left or right successor (depending on the value of the condition), then removed the predecessor from the successor.

the instruction is then removed using eraseFromParent

```
if (value->getZExtValue() != 0) {
    // Create here
    auto newBr = llvm::BranchInst::Create(b->getSuccessor(0), b);
    b->getSuccessor(1)->removePredecessor(b->getParent());
} else {
    auto newBr = llvm::BranchInst::Create(b->getSuccessor(1), b);
    b->getSuccessor(0)->removePredecessor(b->getParent());
}
b->eraseFromParent();
```

DeadBlock

First in getAnalysisUsage(), we included that the ConstantBranch pass was required before this pass.

```
// PA6: Implement
Info.addRequired<ConstantBranch>();
```

Instead of using the depthfirstIterator, I simply implemented the postOrder traversal function from my PA4 implementation as PostOrderTraversal is also depth first search.

```
void postOrder(BasicBlock *currBlock, std::set<BasicBlock *>& visited) {
    if (currBlock == nullptr) {
        return;
    }

    visited.insert(currBlock);

    for (succ_iterator sit = succ_begin(currBlock); sit != succ_end(currBlock); sit++) {
        if (visited.find(*sit) == visited.end()) {
            postOrder(*sit, visited);
        }
    }

    return;
}
```

The rest was done according to the handout, where we check all the basic blocks in F, and if not in the visited set then we can determine it unreachable. From there, we iterate over each unreachable block and for all the successors of a unreachable block, we remove the block as the successors' predecessor

```
for (succ_iterator sit = succ_begin(b); sit != succ_end(b); sit++) {
    sit->removePredecessor(b);
}
b->eraseFromParent();
```

LICM

For LICM, again we closely followed the handout, and added three new member functions, first one to check if an instruction is hoistable, and then to hoist individual instructions, and finally an overall hoisting of all the instructions for all necessary loops.

Because of so many additional information required:

```
Info.setPreservesCFG();
Info.addRequired<DeadBlocks>();
Info.addRequired<DominatorTreeWrapperPass>();
Info.addRequired<LoopInfo>();
```

For isSafetoHoist, we simply checked all the necessary condition for a instruction to be deemed hoistable.

```
if ((mCurrLoop->hasLoopInvariantOperands(inst))
    && (isSafeToSpeculativelyExecute(inst))
    && ((isa<BinaryOperator>(inst)
            || isa<CastInst>(inst)
            || isa<SelectInst>(inst)
            || isa<GetElementPtrInst>(inst)
            || isa<CmpInst>(inst)))) {
    isSafe = true;
}

return isSafe;
```

Hoisting the individual instruction was also simple affair as the basic block relevant is simply the block right before the header of the loop and in that block, we get the last instruction (which is likely a branch) then insert the instruction right before the terminator.

Finally, the hoistPreorder is the recursive runOnLoop that implements the LICM. We check if the current node is part of the current loop, and if so, we check all the instructions in it and check if its safe to hoist, then hoist it out.

```
if (isSafeToHoistInstr(currInst)) {
    hoistInstr(currInst);
}
```

Here, it's important to note that we need to implement the incrementation after saving the current instruction,

```
for (auto it = bb->begin(); it != bb->end();) {
    llvm::Instruction *currInst = it;
    // increment here
    it++;;
```

Finally, this is the recursive portion of the function in preorder,

```
auto children = node->getChildren();
for (auto child : children) {
    hoistPreOrder(child);
}
```