Aditya Bhandaru
Zhe Qian
15-745

1. Writeup Report

FunctionInfo:
Basically, we used llvm Function class's public member functions to get attributes like *name, number of arguments and blocks*. As to the *number of instructions,* we set a counter and iterate through all basic blocks to sum up there instructions. The hardest part of this is to get the number of calls. We figured out that it's impossible to get what we want without go through all the instructions in the module. So we create a new function with both current Function and Module as arguments. And we check if an instruction can be casted to a call instruction and if the function called equals to current function. In that way, we get the number of calls of each function.

LocalOpts:
For local optimizations, we look at each function and block individually before apply any transformations. For strength reduction, we look for multiplication or division operations by powers of 2, and covert them into shifts. To identify algebraic identities, we inspect each operand of all binary operator instructions. If they form an identify, we replace the instruction with the appropriate constant or expression. Finally, for constant folding, we first bypass loads directly from stores to expose constants out of memory. Then we look for binary operations on constants and reduce them until no more can be found. A lot of optimizations are supported, far beyond the scope of the assignment (2 each), but not necessarily exhaustive.

2. List of source code

FunctionInfo.cpp

```
// 15-745 S14 Assignment 1: FunctionInfo.cpp
// Group: bovik, bovik2
/////////////////////////////////////////////////////////////////
///////////////

#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/Instructions.h"

#include <ostream>
#include <fstream>
#include <iostream>

// useful headers
using namespace llvm;
using std::cout;
using std::endl;

namespace {

class FunctionInfo : public ModulePass {

//
// Private helper functions
//

 private:
  void getFunctionInfo(Module& module, Function& function) {
    // useful information
    iplist<BasicBlock>& blocks = function.getBasicBlockList();

    // determine all quantities we need.
    bool is_var_arg = function.isVarArg();
    size_t arg_count = function.arg_size();
    size_t callsite_count = getCallCount(module, function);
    size_t block_count = blocks.size();
    size_t instruction_count = 0;
    for (iplist<BasicBlock>::iterator it = blocks.begin(); it !=
blocks.end(); ++it) {
      BasicBlock& block = *it;
      instruction_count += block.getInstList().size();
    }

    // output in specified format
    cout << function.getName().data() << ",\t";
    if (is_var_arg) {
      cout << "*,\t";
```

```
    } else {
      cout << arg_count << ",\t";
    }
    cout << callsite_count << ",\t";
    cout << block_count << ",\t";
    cout << instruction_count << endl;
  }

  size_t getCallCount(Module& module, Function& function) {
    size_t count = 0;
    for (Module::iterator fn = module.begin(); fn !=
module.end(); ++fn) {
      iplist<BasicBlock>& blocks = (*fn).getBasicBlockList();

      for (iplist<BasicBlock>::iterator it = blocks.begin(); it
!= blocks.end(); ++it) {
        iplist<Instruction>& instructions = (*it).getInstList();

        for (iplist<Instruction>::iterator itr =
instructions.begin(); itr != instructions.end(); ++itr) {
          Instruction* instr = &(*itr);
          if (CallInst* call = dyn_cast<CallInst>(instr)) {
            if (call->getCalledFunction() == &function)
              count++;
          }
        }
      }
    }

    // clean up
    return count;
  }

  // Output the function information to standard out.
  // This function name makes no sense.
  void printFunctionInfo(Module& M) { }


public:

  //
  // Generic setup stuff
  //
  static char ID;
  FunctionInfo() : ModulePass(ID) { }
  ~FunctionInfo() { }

  // We don't modify the program, so we preserve all analyses
  virtual void getAnalysisUsage(AnalysisUsage &AU) const {
    AU.setPreservesAll();
  }
```

```
  //
  // Assignment code
  //
  virtual bool runOnFunction(Function &F) {

    // always return false
    return false;
  }

  virtual bool runOnModule(Module& M) {
    std::cout << "Module " << M.getModuleIdentifier().c_str() <<
std::endl;
    std::cout << "Name,\tArgs,\tCalls,\tBlocks,\tInsns\n";

    // iterate through all functions in the module
    for (Module::iterator MI = M.begin(), ME = M.end(); MI != ME;
++MI) {
      getFunctionInfo(M, *MI);
    }

    // always return false in this example
    return false;
  }

};

// LLVM uses the address of this static member to identify the
pass, so the
// initialization value is unimportant.
char FunctionInfo::ID = 0;
RegisterPass<FunctionInfo> X("function-info", "15745: Function
Information");

}
```

LocalOpts.cpp

```
#include "LocalOpts.h"

// useful headers
using namespace llvm;
using std::cout;
using std::endl;

namespace local {

void LocalOpts::constantFolding(BasicBlock& block) {
  LLVMContext& context = block.getContext();
  ValueMap<Value*, Value* > lastStore;

  // first pass to propagate stores to loads
```

```
  for (BasicBlock::iterator it = block.begin(); it !=
block.end(); ++it) {
    Instruction* instr = &(*it);
    if (StoreInst* store = dyn_cast<StoreInst>(instr)) {
      Value* pointer = store->getPointerOperand();
      Value* value = store->getValueOperand();
      std::pair<Value*, Value* > pair(pointer, value);
      lastStore.erase(pointer);
      lastStore.insert(pair);
    } else if (LoadInst* load = dyn_cast<LoadInst>(instr)) {
      Value* pointer = load->getPointerOperand();
      Value* value = lastStore.lookup(pointer);
      if (value) {
        ReplaceInstWithValue(block.getInstList(), it, value);
        fold++;
      }
    }
  }

  // second pass to fold some exposed constants
  bool changed = true;
  while (changed) {
    changed = false;
    for (BasicBlock::iterator it = block.begin(); it !=
block.end(); ++it) {
      Instruction* instr = &(*it);
      if (BinaryOperator* binOp =
dyn_cast<BinaryOperator>(instr)) {
        BinaryOperator::BinaryOps opcode = binOp->getOpcode();
        ConstantInt* left = dyn_cast<ConstantInt>(binOp-
>getOperand(0));
        ConstantInt* right = dyn_cast<ConstantInt>(binOp-
>getOperand(1));
        // compress if both constant with correct operation
        if (left && right) {
          uint64_t leftVal = left->getValue().getZExtValue();
          uint64_t rightVal = right->getValue().getZExtValue();
          if (opcode == Instruction::Add) {
            ConstantInt* value =
ConstantInt::get(Type::getInt32Ty(context), leftVal + rightVal);
            ReplaceInstWithValue(block.getInstList(), it, value);
            changed = true;
            fold++;
          } else if (opcode == Instruction::Sub) {
            ConstantInt* value =
ConstantInt::get(Type::getInt32Ty(context), leftVal - rightVal);
            ReplaceInstWithValue(block.getInstList(), it, value);
            changed = true;
            fold++;
          } else if (opcode == Instruction::Mul) {
            ConstantInt* value =
ConstantInt::get(Type::getInt32Ty(context), leftVal * rightVal);
            ReplaceInstWithValue(block.getInstList(), it, value);
```

```
            changed = true;
            fold++;
          } else if (opcode == Instruction::SDiv) {
            ConstantInt* value =
ConstantInt::get(Type::getInt32Ty(context), leftVal / rightVal);
            ReplaceInstWithValue(block.getInstList(), it, value);
            changed = true;
            fold++;
          }
        }
      }
    }
  }
}

void LocalOpts::strengthReduction(BasicBlock& block) {
  LLVMContext& context = block.getContext();

  for (BasicBlock::iterator it = block.begin(); it !=
block.end(); ++it) {
    Instruction* instr = &(*it);
    if (BinaryOperator* binOp = dyn_cast<BinaryOperator>(instr))
{
      BinaryOperator::BinaryOps opcode = binOp->getOpcode();
      Value* left = binOp->getOperand(0);
      Value* right = binOp->getOperand(1);

      // determine the operand types
      Instruction* leftInstr = dyn_cast<Instruction>(left);
      Instruction* rightInstr = dyn_cast<Instruction>(right);
      ConstantInt* leftValue = dyn_cast<ConstantInt>(left);
      ConstantInt* rightValue = dyn_cast<ConstantInt>(right);

      // multiply instruction
      if (opcode == Instruction::Mul) {
        if (leftInstr && rightValue) {
          uint64_t value = rightValue->getValue().getZExtValue();
          uint64_t log2Value = log2(value);
          if (value == (1 << log2Value)) {
            ConstantInt* amount =
ConstantInt::get(Type::getInt32Ty(context), log2Value);
            Instruction* shift =
BinaryOperator::Create(Instruction::Shl, leftInstr, amount);
            ReplaceInstWithInst(block.getInstList(), it, shift);
            strength++;
          }
        } else if (leftValue && rightInstr) {
          uint64_t value = leftValue->getValue().getZExtValue();
          uint64_t log2Value = log2(value);
          if (value == (1 << log2Value)) {
            ConstantInt* amount =
ConstantInt::get(Type::getInt32Ty(context), log2Value);
```

```
            Instruction* shift =
BinaryOperator::Create(Instruction::Shl, rightInstr, amount);
            ReplaceInstWithInst(block.getInstList(), it, shift);
            strength++;
          }
        }

      // divide instruction
      } else if (opcode == Instruction::SDiv) {
        if (leftInstr && rightValue) {
          uint64_t value = rightValue->getValue().getZExtValue();
          uint64_t log2Value = log2(value);
          if (value == (1 << log2Value)) {
            ConstantInt* amount =
ConstantInt::get(Type::getInt32Ty(context), log2Value);
            Instruction* shift =
BinaryOperator::Create(Instruction::LShr, leftInstr, amount);
            ReplaceInstWithInst(block.getInstList(), it, shift);
            strength++;
          }
        }
      }
    }
  }
}

void LocalOpts::algebraicIdentities(BasicBlock& block) {
  LLVMContext& context = block.getContext();

  // iterate through instructions
  for (BasicBlock::iterator it = block.begin(); it !=
block.end(); ++it) {
    Instruction* instr = &(*it);
    if (BinaryOperator* binOp = dyn_cast<BinaryOperator>(instr))
{
      BinaryOperator::BinaryOps opcode = binOp->getOpcode();
      Value* left = binOp->getOperand(0);
      Value* right = binOp->getOperand(1);

      // determine the operand types
      Instruction* leftInstr = dyn_cast<Instruction>(left);
      Instruction* rightInstr = dyn_cast<Instruction>(right);
      ConstantInt* leftValue = dyn_cast<ConstantInt>(left);
      ConstantInt* rightValue = dyn_cast<ConstantInt>(right);

      // both sources are instructions
      if (leftInstr && rightInstr) {
        if (leftInstr->isSameOperationAs(rightInstr)) {
          if (opcode == Instruction::Sub) {
            ConstantInt* value =
ConstantInt::get(Type::getInt32Ty(context), 0);
            ReplaceInstWithValue(block.getInstList(), it, value);
            algebra++;
```

```
          } else if (opcode == Instruction::SDiv) {
            // TODO: does not catch divide by zero
            ConstantInt* value =
ConstantInt::get(Type::getInt32Ty(context), 1);
            ReplaceInstWithValue(block.getInstList(), it, value);
            algebra++;
          }
        }
        // clean up references if possible
        if (leftInstr->use_empty())
          leftInstr->eraseFromParent();
        if (rightInstr->use_empty())
          rightInstr->eraseFromParent();
      }

      // left source is a constant
      else if (leftInstr && rightValue) {
        if ((opcode == Instruction::Mul || opcode ==
Instruction::SDiv) &&
             rightValue->isOne()) {
          ReplaceInstWithValue(block.getInstList(), it,
leftInstr);
          algebra++;
        } else if ((opcode == Instruction::Add || opcode ==
Instruction::Sub) &&
                    rightValue->isZero()) {
          ReplaceInstWithValue(block.getInstList(), it,
leftInstr);
          algebra++;
        }
      }

      // right source is a constant
      else if (leftValue && rightInstr) {
        if (opcode == Instruction::Mul && leftValue->isOne()) {
          ReplaceInstWithValue(block.getInstList(), it,
rightInstr);
          algebra++;
        } else if (opcode == Instruction::Add && leftValue-
>isZero()) {
          ReplaceInstWithValue(block.getInstList(), it,
rightInstr);
          algebra++;
        }
      }
    }
  }
}

bool LocalOpts::runOnModule(Module& module) {
  // init counters
  strength = 0;
  fold = 0;
```

```
  algebra = 0;

  // run over functions
  for (Module::iterator it = module.begin(); it != module.end();
++it) {
    eachFunction(*it);
  }

  // print out tranform counts
  cout << "Transformations applied:" << endl;
  cout << "  Algebraic Identities: " << algebra << endl;
  cout << "  Constant Folding: " << fold << endl;
  cout << "  Strength Reductions: " << strength << endl;
  return false;
}

void LocalOpts::eachFunction(Function& function) {
  for (Function::iterator it = function.begin(); it !=
function.end(); ++it) {
    algebraicIdentities(*it);
    strengthReduction(*it);
    constantFolding(*it);
  }
}

// Changed so we can actually modify the code tree.
void LocalOpts::getAnalysisUsage(AnalysisUsage &AU) const {
  AU.setPreservesCFG();
}

uint64_t LocalOpts::log2(uint64_t x) {
  int i = 0;
  while (x >>= 1) {
    i++;
  }
  return i;
}

// LLVM uses the address of this static member to identify the
pass, so the
// initialization value is unimportant.
char LocalOpts::ID = 0;
RegisterPass<LocalOpts> X("local-opts", "15745: Local
Optimizations");

}
```

3. Additional test case()

small.c

```
int compute (int a, int b, int c)
{
  int result = (a/a);

  a = a + 0;
  b = b - 0;
  c = 1 * c;

  result *= (b/b);
  result += (b-b);
  result /= result;
  result -= result;
  return result;
}
```

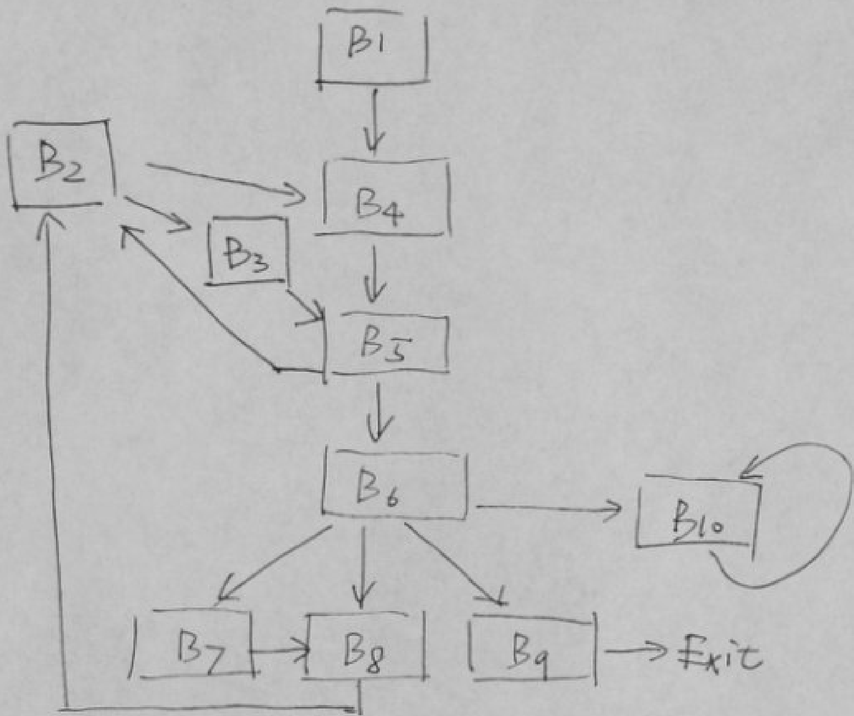
Expected output:

```
; Function Attrs: nounwind
define i32 @compute(i32 %a, i32 %b, i32 %c) #0 {
entry:
  %a.addr = alloca i32, align 4
  %b.addr = alloca i32, align 4
  %c.addr = alloca i32, align 4
  %result = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  store i32 %b, i32* %b.addr, align 4
  store i32 %c, i32* %c.addr, align 4
  store i32 1, i32* %result, align 4
  store i32 %a, i32* %a.addr, align 4
  store i32 %b, i32* %b.addr, align 4
  store i32 %c, i32* %c.addr, align 4
  store i32 1, i32* %result, align 4
  store i32 1, i32* %result, align 4
  store i32 1, i32* %result, align 4
  store i32 0, i32* %result, align 4
  ret i32 0
}
```

4. Answer of Section 5 (scanned)

5.1 CFG Basics

```
x = 1
y = 0
goto L2                                     B1

L1. y = x * y
    if (x < 50) goto L2                     B2

y = x - y
goto L3                                     B3

L2: y = x + y                               B4

L3. print (y)
    if (y < 1000) goto L1                   B5

switch (x) | 0 => L6 | 1 => L4 | 101 => L7 | default => L5 |   B6

L4: print ("!")                             B7

L5: x = x - 1
    goto L1                                 B8

L6. return y                                B9

L7. goto L7                                 B10
```



5.2 Available Expression.

BB	EVAL	KILL
1	a=b+c d=b+b e=b*d i=1	φ
2	b=d. f=b+c y=i+1	a=b+c d=b*b e=b*d.
3	c=14 e=b*b	f=b+c
4	d = b×b	b=d. e=b*d
5.	φ	y=i+1

TABLE 1.

5.2 Available Expression (continued)

BB	IN	OUT.
1	ϕ	$a=b+c$ $d=b*b$ $e=b*d$ $i=1$
2	$a=b+c$ $d=b*b$ $e=b*d$ $i=1$	$b=d$ $f=b+c$ $i=1$ $y=i+1$
3	$b=d$ $f=b+c$ $i=1$ $y=i+1$	$b=d$ $c=14$ $e=b*b$ $y=i+1$ $i=1$.
4	$b=d$ $f=b+c$ $i=1$ $y=i+1$	$d=b*b$ $f=b+c$ $i=1$ $y=i+1$
5.	$i=1$ $y=i+1$	ϕ

5.3

(1). set of ~~expressions~~ variables.

(2) bottom-up

(3) $IN[B] = f_B(OUT[B]) = DEF[B] \cup (OUT[B] - USE[B])$

where: $DEF[B]$: sets of variables defined in basic block B;

$USE[B]$: set of locally exposed uses in basic block B;

(4). $\wedge = \cup$ $\quad OUT[B] = \cap\, IN[B]$.

(5) $IN[Exit] = \phi$ $\quad$ (6) $IN[B] = Univ.$

(7) Yes. REVERSED-ORDER. Need to anaylzed from bottom to up.

(8). Yes Basically. numbers of defs in a program is finite.

(9). $IN[Exit] = \phi$.

For each basic block B other than Exit.

$IN[B] = Univ.$

while (changes to any IN[] occur) {

for each basic block B other than Exit {

$OUT[B] = \cap\, IN[S]$. for all successors of B.

$IN[B] = f_B(OUT[B])$

}

}

IN[B] of each basic block B indicates. the "faint" variables