Project 3: Liveness Analysis Report

Group member: Zixu Cao, Chuanye Xiong

- **Code summary**

Basically, the algorithm can be divided into three major parts. The first part is to calculate the VarKill and UEVar for every block in the CFG. The second part is iterating for the liveout variables for every block in the CFG until the liveout converge (won't change any more) for every block. The third part is based on the data we already have and print the UEVAR, VARKILL and LIVEOUT in the right format.

- **Video Link**

https://drive.google.com/drive/folders/1A_D9Y0gD5PmNfge7K1Vnt01G0C9LOzjL?usp=share_link

- **Basic data structure**

The major data structure we are using is three hash tables:

1. block_UEVar, in which key is the block name and value is UEVar.
2. block_VarKill, in which key is the block name and value is VarKill.
3. block_LiveOut, in which key is the block name and value is LiveOut.

Each hash table correlating the block name (key is string) with the upward exposed variables (string vector) in the single block, variables killed in the single block(string vector) and Live out variables (string vector) from the single block.

- **Implementation details**

```
for (auto& inst : basic_block)
{
    if(inst.getOpcode() == Instruction::Load){
        string var = string(inst.getOperand(0)->getName());

        // For load situation if we can't fina a var in VarKill or UEVar
// we push it to the vector of UeVar
        if(std::find(VarKill.begin(), VarKill.end(), var) == VarKill.end()\
        && std::find(UEVar.begin(), UEVar.end(), var) == UEVar.end()){
            UEVar.push_back(var);
        }
    }
```

Figure 1. code part1

The code part 1 (figure 1) aiming at calculate VarKill and UEVar for every block. We traverse all the block, and in each block, we traverse each instruction.
If the instruction is loading instruction: then we following the algorithm(Figure 2), provided in the ppt. if the var not in the VarKill set and also not in the UEVar set. We insert it in the UEVar set.

Compute ***UEVAR(N)*** and ***VARKILL(N)***

assume block N has operations o_1, o_2, …, o_k

VARKILL(N) = Ø
UEVar(N) = Ø

FOR i = 1 to k
  assume o_i *is "x ← y op z"*
  IF y ∉ **VARKILL**(N)
    UEVar(N) = **UEVAR**(N) ∪ y
  IF z ∉ **VARKILL**(N)
    UEVar(N) = **UEVAR**(N) ∪ z
  VARKILL(N) = **VARKILL**(N) ∪ x

- **Complexity**: O(k)

Figure 2. the algorithm for compute UEVar and VarKill set

```
        if(inst.getOpcode() == Instruction::Store){
//store operation part, caculate for VarKill
            string var1 = "";
            string var2 = "";
            // if first operand is a constant, ignore it useless
            if(isa<ConstantInt>(inst.getOperand(0))){
                var1 = "";
            }
            else{
                //if first operand is a binary op
                var1 = string(dyn_cast<User>(dyn_cast<User>(inst.getOperand(0))\
                ->getOperand(0))->getOperand(0)->getName());
                var2 = string(dyn_cast<User>(dyn_cast<User>(inst.getOperand(0))\
                ->getOperand(1))->getOperand(0)->getName());
                // if first operand returns empty so it may a register or load
                if(string(inst.getOperand(0)->getName()) == ""){
                    var1 = string(dyn_cast<User>(inst.getOperand(0))->getOperand(0)->getName());
                }
                // if var1 is a constant
                if(isa<ConstantInt>(dyn_cast<User>(dyn_cast<User>(inst.getOperand(0))\
                    ->getOperand(0))->getOperand(0))){
                    var1 = "";
                }
    }
    else{
        //if first operand is a binary op
        var1 = string(dyn_cast<User>(dyn_cast<User>(inst.getOperand(0))\
        ->getOperand(0))->getOperand(0)->getName());
        var2 = string(dyn_cast<User>(dyn_cast<User>(inst.getOperand(0))\
        ->getOperand(1))->getOperand(0)->getName());
        // if first operand returns empty so it may a register or load
        if(string(inst.getOperand(0)->getName()) == ""){
            var1 = string(dyn_cast<User>(inst.getOperand(0))->getOperand(0)->getName());
        }
        // if var1 is a constant
        if(isa<ConstantInt>(dyn_cast<User>(dyn_cast<User>(inst.getOperand(0))\
            ->getOperand(0))->getOperand(0))){
            var1 = "";
        }

        // if var2 is a constant
        if(isa<ConstantInt>(dyn_cast<User>(dyn_cast<User>(inst.getOperand(0))\
            ->getOperand(1))->getOperand(0))){
            var2 = "";
        }
    }
```

Figure 3. code part2

As is shown in Figure3, If the instruction is store, var1, var2 all has chance to be the real variable. So we follow the logic above (in Figure 2) to insert the variable to the UEVar set.

```
            string var_name = string(inst.getOperand(1)->getName());

            // if var1 is not in VarKill set or UEVar set push it
            if(std::find(VarKill.begin(), VarKill.end(), var1) == VarKill.end() \
            && std::find(UEVar.begin(), UEVar.end(), var1) == UEVar.end()){
                UEVar.push_back(var1);
            }

            /* if var2 is not in VarKill set or  UEVar set */
            if(std::find(VarKill.begin(), VarKill.end(), var2) == VarKill.end() \
            && std::find(UEVar.begin(), UEVar.end(), var2) == UEVar.end()){
                UEVar.push_back(var2);
            }

            // if var is not already in VarKill push it
            if(std::find(VarKill.begin(), VarKill.end(), var_name) == VarKill.end()){
                VarKill.push_back(var_name);
            }
        }

    } //end for inst
```

Figure 4. code part3

As we can see in the Figure 4, var_name is the element that we must killed. So we check if it's not in the VarKill set, we push it into the VarKill set. For example: as for the instruction x← y op z. var1 is the y, var2 is the z and var_name is x.

```
        // Save sets to related block
        block_UEVar.insert(make_pair(block_name, UEVar));
        block_VarKill.insert(make_pair(block_name, VarKill));

    } // end for block
```

Figure 5. code part4

For each basic block, we get the data of the UEVar set and the VarKill set and save them to the corresponding hash table. (block_UEVar and block_VarKill). Inside the hash table, the key is the block name and value is the UEVar set or VarKill set.

Next we will do the iterations to get the LiveOut set for every block based on the VarKill and UEVar we already have.

```
// initialize set for LiveOut
for (auto& basic_block : F){
    vector<string> emptySet = {};
    block_LiveOut.insert(make_pair(string(basic_block.getName()), emptySet));
}
```

Figure 6. code part5

Before the iteration, we initialize the LiveOut set for every basic block to be empty set, as is shown in Figure 6.

We follow the algorithm told in the class, as shown in the Figure 6 to iterate the LiveOut set for every basic block. Although go through all the blocks in the CFG is not the most efficient method for the liveout analysis, but it's a useful and simple way to solve the task.

```
FOR block N in CFG
  LIVEOUT(N) = Ø
continue = true
WHILE(continue)
  continue = false
  FOR block N in CFG
    LIVEOUT(N) = ∪X∈succ(N)(LIVEOUT(X)-VARKILL(X)∪UEVar(X))
    IF LIVEOUT(N) changed
          continue = true
```

Initialization

Inefficiency ?

Figure 7. algorithm for iterating the Liveout set for each basic block

```
// initialize set for LiveOut
for (auto& basic_block : F){
    vector<string> emptySet = {};
    block_LiveOut.insert(make_pair(string(basic_block.getName()), emptySet));
}

// Compute liveout var by
unordered_map<string ,std::vector<string>>::const_iterator block_find;
bool cont = true;
while(cont){
    cont = false;
    for (auto& basic_block : F){
        std::vector<string> liveOut;
        std::vector<string> liveOut_temp;
        std::vector<string> liveOut_succ;
        std::vector<string> VarKill_succ;
        std::vector<string> UEVar_succ;
        std::vector<string> union_succ;

        for(BasicBlock *succ : successors(&basic_block)){
            std::vector<string> diff_temp;
            std::vector<string> union_temp;

            // check if the Liveout can be find of current block
            block_find = block_LiveOut.find(string(succ->getName()));
            liveOut_succ = block_find->second;
            block_find = block_VarKill.find(string(succ->getName()));
            VarKill_succ = block_find->second;
            block_find = block_UEVar.find(string(succ->getName()));
            UEVar_succ = block_find->second;

            // compute LiveOut(X) - VarKill(X)
            std::set_difference(liveOut_succ.begin(), liveOut_succ.end(),\
            VarKill_succ.begin(), VarKill_succ.end(), std::back_inserter(diff_temp));
            std::set_union(diff_temp.begin(), diff_temp.end(),\
            UEVar_succ.begin(), UEVar_succ.end(), std::back_inserter(union_temp));

            for(auto it: union_temp){
                union_succ.push_back(it);
            }
        }
    }
```

Figure 8. code part 6

In the code part 6 above we first set continue iteration to be true. The ending iteration criterion is that the LiveOut set for all the block won't change any more. the while(cont) is

to tell the iteration proceed or not. The outer for loop is traverse for all the basic block in the CFG and the inner for loop is traverse for the all the successor blocks for a certain block in the CFG. For each successor, we compute the (LiveOut(X) – VarKill(X)∪UEVar(X)).

As is shown in Figure 9, After we traverse all the successors, we take the union $U_{X \in successor}$ (LiveOut(X) – VarKill(X)∪UEVar(X)), which would be the LiveOut set for certain block N in current iteration. We save that into the union_succ in the code.

```
            // add UEVar(X) to liveout
            std::set_union(diff_temp.begin(), diff_temp.end(),\
            UEVar_succ.begin(), UEVar_succ.end(), std::back_inserter(union_temp));

            for(auto it: union_temp){
                union_succ.push_back(it);
            }

        }

        // Compute Union of Successors
std::sort(union_succ.begin(), union_succ.end());
        union_succ.erase(std::unique(union_succ.begin(),\
            union_succ.end()), union_succ.end());
```

Figure 9. code part 7

We get the previous iteration Liveout set data from the hash table (liveOut in the code), and compare with the current step LiveOut set data (union_succ). If its not the same, we continue iteration while If it's the same, we end the iteration. (Figure 10)

```
            // retrieve current LiveOut
            block_find = block_LiveOut.find(string(basic_block.getName()));
            liveOut = block_find->second;

            // If LiveOut(N) is changed mark it
            if(liveOut != union_succ){
                cont = true;
            }

            // Update LiveOut
            auto it = block_LiveOut.find(string(basic_block.getName()));
            it->second = union_succ;


        }
    }
```

Figure 10. code part 8

In the end, we update the current LiveOut set data and save it into the hash table, as is shown in the figure 11.

```
        // Update LiveOut
        auto it = block_LiveOut.find(string(basic_block.getName()));
        it->second = union_succ;


    }
}
```

Figure 11. code part 9

When the iteration is done, we get the LiveOut set data for every block and saved that into the hash table.

The last part of our code is just retrieve the data from the hash table and print out the data. We traverse through all the block, for each block, we print out the block name, UEVAR set, VARKILL set and LIVEOUT set in the right format, as is shown in the Figure 12.

```
for (auto& basic_block : F){
    std::vector<string> UEVar_temp;
    std::vector<string> VarKill_temp;
    std::vector<string> LiveOut_temp;
    unordered_map<string ,std::vector<string>>::const_iterator block_f

    block_find = block_UEVar.find(string(basic_block.getName()));
    UEVar_temp = block_find->second;
std::sort(UEVar_temp.begin(), UEVar_temp.end());
    errs() << "------ " << basic_block.getName() << " ------\n";
    errs() << "UEVAR: ";
    for(auto a: UEVar_temp){
        errs() << a << " ";
    }
    errs() << "\n";

    block_find = block_VarKill.find(string(basic_block.getName()));
    VarKill_temp = block_find->second;
std::sort(VarKill_temp.begin(), VarKill_temp.end());
    errs() << "VARKILL: ";
    for(auto a: VarKill_temp){
        errs() << a << " ";
    }
    errs() << "\n";

    block_find = block_LiveOut.find(string(basic_block.getName()));
    LiveOut_temp = block_find->second;
std::sort(LiveOut_temp.begin(), LiveOut_temp.end());
    errs() << "LIVEOUT: ";
    for(auto a: LiveOut_temp){
        errs() << a << " ";
    }
    errs() << "\n";
}
```

Figure 11. code part 10

- **Implementation results**

1) For test1.c, we get the output:

```
(base) cxiong@ucr-secure-23-10-13-140-130 test % opt -load-pass-plugin ../Pass/build/libLLVMValueNumberingPass.so  -passes=value-numbering test1.ll
WARNING: You're attempting to print out a bitcode file.
This is inadvisable as it may cause display problems. If
you REALLY want to taste LLVM bitcode first-hand, you
can force output with the `-f' option.

ValueNumbering: test
------ entry ------
UEVAR: b c
VARKILL: e
LIVEOUT:  a b c e
------ if.then ------
UEVAR:  a
VARKILL: e
LIVEOUT: c e
------ if.else ------
UEVAR: b c
VARKILL: a
LIVEOUT: c e
------ if.end ------
UEVAR: c e
VARKILL: a
LIVEOUT:
(base) cxiong@ucr-secure-23-10-13-140-130 test %
```

2) For test2.c, we get the output:

```
(base) cxiong@ucr-secure-23-10-13-140-130 test % opt -load-pass-plugin ../Pass/build/libLLVMValueNumberingPass.so  -passes=value-numbering test2.ll
WARNING: You're attempting to print out a bitcode file.
This is inadvisable as it may cause display problems. If
you REALLY want to taste LLVM bitcode first-hand, you
can force output with the `-f' option.

ValueNumbering: test
------ entry ------
UEVAR:
VARKILL: a c
LIVEOUT:  a b d e
------ do.body ------
UEVAR:  a b
VARKILL: c
LIVEOUT:  b c d e
------ if.then ------
UEVAR:  c d
VARKILL: c f
LIVEOUT:  b d e
------ if.else ------
UEVAR:  d e
VARKILL: a e
LIVEOUT:  b d e
------ if.end ------
UEVAR:  b
VARKILL: a
LIVEOUT:  a b d e
------ do.cond ------
UEVAR: a
VARKILL:
LIVEOUT:  a b d e
------ do.end ------
UEVAR:  a
VARKILL: a
LIVEOUT:
```

3) For test3.c, we get the output:

```
(base) cxiong@ucr-secure-23-10-13-140-130 test % opt -load-pass-plugin ../Pass/build/libLLVMValueNumberingPass.so  -passes=value-numbering test3.ll
WARNING: You're attempting to print out a bitcode file.
This is inadvisable as it may cause display problems. If
you REALLY want to taste LLVM bitcode first-hand, you
can force output with the `-f' option.

ValueNumbering: test
------ entry ------
UEVAR:
VARKILL: a b i
LIVEOUT:  a b c d e i
------ for.cond ------
UEVAR: i
VARKILL:
LIVEOUT:  a b c d e i
------ for.body ------
UEVAR: a d
VARKILL: c e
LIVEOUT:  a b c d e i
------ for.inc ------
UEVAR:  i
VARKILL: i
LIVEOUT:  a b c d e i
------ for.end ------
UEVAR: d e
VARKILL:
LIVEOUT: b c d e
------ if.then ------
UEVAR: c d
VARKILL: f
LIVEOUT:
------ if.else ------
UEVAR: b e
VARKILL: e
LIVEOUT:
------ if.end ------
UEVAR:
VARKILL:
LIVEOUT:
```

4) For test4.c, we get the output:

```
(base) cxiong@ucr-secure-23-10-13-140-130 test % opt -load-pass-plugin ../Pass/build/libLLVMValueNumberingPass.so  -passes=value-numbering test4.ll
WARNING: You're attempting to print out a bitcode file.
This is inadvisable as it may cause display problems. If
you REALLY want to taste LLVM bitcode first-hand, you
can force output with the `-f' option.

ValueNumbering: test
------ entry ------
UEVAR:
VARKILL: a b c e i
LIVEOUT:  a b e i
------ for.cond ------
UEVAR: i
VARKILL:
LIVEOUT:  a b e i
------ for.body ------
UEVAR: a b
VARKILL: c
LIVEOUT:  a c e i
------ while.cond ------
UEVAR: e i
VARKILL:
LIVEOUT:  a c e i
------ while.body ------
UEVAR: a i
VARKILL: i
LIVEOUT:  a c e i
------ while.end ------
UEVAR: a c
VARKILL: b
LIVEOUT:  a b e i
------ for.inc ------
UEVAR:  i
VARKILL: i
LIVEOUT:  a b e i
------ for.end ------
UEVAR:
VARKILL:
LIVEOUT:
```

In summary, we have successfully the Liveness analysis for all the test case.