Compiler Construction Project Phase 3

Name: Paikana Suhas Rao **SID:862394626**

Video Link: https://drive.google.com/file/d/1A4bJZF3Y-EKyabWJ5vEcLar5IIQAZh8Q/view?usp=share_link

Liveness Analysis:

A variable is live if it holds a value that will/might be used in the future. The representation of the program that we use for liveness analysis (determining which variables are live at each point in a program), is a control flow graph. The nodes in a control flow graph are basic statements (instructions). There is an edge from statement x to statement y if x can be immediately followed by y (control flows from x to y).

Basic data-flow analysis pass:

Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program. A program's control-flow graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate. The information gathered is often used by compilers when optimizing a program. A canonical example of a data-flow analysis is reaching definitions.

LLVM Compiler Infrastructure

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. Despite its name, LLVM has little to do with traditional virtual machines. The name "LLVM" itself is not an acronym; it is the full name of the project.

LLVM began as a research project at the University of Illinois, with the goal of providing a modern, SSA-based compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages. Since then, LLVM has grown to be an umbrella project consisting of a number of subprojects, many of which are being used in production by a wide variety of commercial and open source projects as well as being widely used in academic research. Code in the LLVM project is licensed under the "Apache 2.0 License with LLVM exceptions"

Algorithm Approach:

1. At first, we'll through the instructions in a particular basic block and add it to the VARKILL when it used on the left-hand side of the instruction. This means that this variable might get changed in this basic block and the previous value which it was receiving might not be forwarded above because it might get change.
2. Next, we will fill the UEVAR Map. For this we'll go through each of the basic block and pick out the variable which are on the right-hand side of the instruction. And if it part of the basic block's VARKILL then we'll skip this as this will get killed in the process anyways. This is all stored in a set of variables as we don't want the repeats of the variables anyways.
3. For better processing of the fields, we'll use a PREDBBMAP keep hold of all the predecessor for all the basic blocks.
4. In the next step we'll use a iterative procedure to fill out the LIVEOUT basic map for each of the basic block. So, this calculated on the depending on the predecessors LIVEOUT for each of the basic block.
5. Once we have all the three maps available we can go through the three maps for each of the basic blocks and errs() the values.

Output Sample:

```
prao013@ITSloan-PF1T1J68:~/.ssh/data-flow-analyzer-suhas-rao/test$ opt -load-pass-pl
mberingPass.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.


-----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:
```

Source Code:

Available in: https://github.com/CS201-Winter23/data-flow-analyzer-suhas-rao/blob/main/Pass/Transforms/ValueNumbering/ValueNumbering.cpp

```
#include
"llvm/IR/LegacyPassManager.
h"
                                #include "llvm/Passes/PassPlugin.h"
                                #include "llvm/Passes/PassBuilder.h"
                                #include "llvm/IR/PassManager.h"
                                #include <string>
                                #include <vector>
                                #include <set>
                                #include <map>
                                #include "llvm/Support/raw_ostream.h"
                                #include "llvm/IR/CFG.h"
                                #include <algorithm>
                                #include <iterator>
                                using namespace std;


                                using namespace llvm;


                                namespace {
                                void getunion(set<string> a, set<string> b)
                                       {
                                       a.insert(b.begin(), b.end());
                                       }
                                // This method implements what the pass does
                                void visitor(Function &F){


                                    // Here goes what you want to do with a pass

                                               string func_name = "main";

                                           // Comment this line
```

```
        //if (F.getName() != func_name) return;
        int count=0;
        map<string, set<string> > PREDBBMAP;

        for (auto& basic_block : F)
        {count++;
        }

        string bbs[count];
        for (auto& basic_block : F)
        {count--;
        bbs[count]=basic_block.getName().str();
        }

        for (auto& basic_block : F)
        {count++;
        }

        /*for(int i=0;i<count;i++){
        errs()<<bbs[i]<<" ";
        }*/

        for (auto& basic_block : F){

        for (auto it = pred_begin(&basic_block), et =
pred_end(&basic_block); it != et; ++it){
                BasicBlock* predecessor = *it;
        PREDBBMAP[basic_block.getName().str()].insert(predeces
sor->getName().str());

        }}


        map<string, set<string> > UEVAR;
        map<string, set<string> > VARKILL;
        for (auto& basic_block : F)
        {
        for (auto& inst : basic_block)
            {
                if(inst.getOpcode() == Instruction::Store){
                    string operand2=inst.getOperand(1)-
>getName().str();
```

```
VARKILL[basic_block.getName().str()].insert(operand2);
                }
                if(inst.getOpcode() == Instruction::Load){
                   string operand1=inst.getOperand(0)-
>getName().str();
                   auto
it=VARKILL[basic_block.getName().str()].find(operand1);
                   if ( it ==
VARKILL[basic_block.getName().str()].end() )

{UEVAR[basic_block.getName().str()].insert(operand1);}
                }
             } // end for inst
       }









       map<string, set<string> > LIVEOUT;
       for(int i=0;i<count;i++){
       LIVEOUT[bbs[i]];
       }

       /*
       for(auto it = LIVEOUT.cbegin(); it != LIVEOUT.cend();
++it)
       {
       errs()<<it->first<<" ";
       }*/

       set<string>::iterator itrr;
       set<string> HOLDER;

       int i=2;
       while(i>0){
       for(int i=0;i<count;i++){
         for (itrr = PREDBBMAP[bbs[i]].begin();
       itrr != PREDBBMAP[bbs[i]].end(); itrr++)
```

```
          {
//     LIVEOUT[*itrr] union ( ( LIVEOUT[bbs[i]] diff
VARKILL[bbs[i]] ) union UEVAR[bbs[i]])
               set_difference(LIVEOUT[bbs[i]].begin(),
LIVEOUT[bbs[i]].end(), VARKILL[bbs[i]].begin(),
VARKILL[bbs[i]].end(),
    std::inserter(HOLDER, HOLDER.end()));

       //     errs()<<"Before"<<"\n";
       //     set<string>::iterator iti;
       //     for (iti = HOLDER.begin(); iti !=
HOLDER.end(); ++iti) {
       //                   errs() << *iti<<" ";
       //     }




       HOLDER.insert(UEVAR[bbs[i]].begin(),UEVAR[bbs[i]].end(
));

       //     errs()<<"After"<<"\n";
       //     set<string>::iterator itf;
       //     for (itf = HOLDER.begin(); itf !=
HOLDER.end(); ++itf) {
       //                   errs() << *itf<<" ";
       //     }


       LIVEOUT[*itrr].insert(HOLDER.begin(),HOLDER.end());

              HOLDER.clear();
       }}
       i--;
       }

       /*
       for(int i=0;i<count;i++){
         for (itrr = PREDBBMAP[bbs[i]].begin();
       itrr != PREDBBMAP[bbs[i]].end(); itrr++)
         {
//     LIVEOUT[*itrr] union ( ( LIVEOUT[bbs[i]] diff
VARKILL[bbs[i]] ) union UEVAR[bbs[i]])
```

```
                set_difference(LIVEOUT[bbs[i]].begin(),
LIVEOUT[bbs[i]].end(), VARKILL[bbs[i]].begin(),
VARKILL[bbs[i]].end(),
    std::inserter(HOLDER, HOLDER.end()));

                errs()<<"Before"<<"\n";
                set<string>::iterator iti;
                for (iti = HOLDER.begin(); iti !=
HOLDER.end(); ++iti) {
                                errs() << *iti<<" ";
                }




        HOLDER.insert(UEVAR[bbs[i]].begin(),UEVAR[bbs[i]].end(
));

                errs()<<"After"<<"\n";
                set<string>::iterator itf;
                for (itf = HOLDER.begin(); itf !=
HOLDER.end(); ++itf) {
                                errs() << *itf<<" ";
                }


        LIVEOUT[*itrr].insert(HOLDER.begin(),HOLDER.end());

                HOLDER.clear();
        }}

        */
```

```
for (auto& basic_block : F)
        {
       errs() <<"\n"<< "-----"<<basic_block.getName()<<"-----
"<<"\n";
//     errs() <<"\n"<<basic_block.getName().str()<<": ";
//             set<string>::iterator itr0;
  // Displaying set elements
//  for (itr0 =
PREDBBMAP[basic_block.getName().str()].begin();
//       itr0 !=
PREDBBMAP[basic_block.getName().str()].end(); itr0++)
//  {
//    errs() << *itr0 << " ";
//  }



           errs() <<"\n"<< "UEVAR:";
              set<string>::iterator itr;

  // Displaying set elements
  for (itr = UEVAR[basic_block.getName().str()].begin();
       itr != UEVAR[basic_block.getName().str()].end();
itr++)
  {
    errs() <<" "<< *itr ;
  }

        //errs()<<UEVAR;
       //    for (int i = 0; i < UEVAR.size(); i++)
       //    {errs() << UEVAR[i] << " ";}

       errs() <<"\n"<< "VARKILL:";
       set<string>::iterator itr1;

  // Displaying set elements
  for (itr1 = VARKILL[basic_block.getName().str()].begin();
       itr1 != VARKILL[basic_block.getName().str()].end();
itr1++)
  {
    errs() <<" "<< *itr1;
  }
 errs() <<"\n"<< "LIVEOUT:";
```

```
        set<string>::iterator itr2;

  // Displaying set elements
  for (itr2 = LIVEOUT[basic_block.getName().str()].begin();
       itr2 != LIVEOUT[basic_block.getName().str()].end();
itr2++)
  {
    errs() << " "<< *itr2 ;
  }

        //err()<<VARKILL;
        //    for (int i = 0; i < VARKILL.size(); i++)
        //    {errs() << VARKILL[i] << " ";}

         } // end for block
        errs()<<"\n";

}




// New PM implementation
struct ValueNumberingPass : public
PassInfoMixin<ValueNumberingPass> {


  // The first argument of the run() function defines on what
level
  // of granularity your pass will run (e.g. Module,
Function).
  // The second argument is the corresponding AnalysisManager
  // (e.g ModuleAnalysisManager, FunctionAnalysisManager)
  PreservedAnalyses run(Function &F, FunctionAnalysisManager
&) {
        visitor(F);
        return PreservedAnalyses::all();



  }

    static bool isRequired() { return true; }
```

```
};
}
```

```
//-----------------------------------------------------------------
// New PM Registration
//-----------------------------------------------------------------
extern "C" ::llvm::PassPluginLibraryInfo LLVM_ATTRIBUTE_WEAK
llvmGetPassPluginInfo() {
  return {
    LLVM_PLUGIN_API_VERSION, "ValueNumberingPass",
LLVM_VERSION_STRING,
    [](PassBuilder &PB) {
      PB.registerPipelineParsingCallback(
        [](StringRef Name, FunctionPassManager &FPM,
        ArrayRef<PassBuilder::PipelineElement>) {
          if(Name == "value-numbering"){
            FPM.addPass(ValueNumberingPass());
            return true;
          }
          return false;
        });
    }};
}
```