Compiler Construction Phase 2 Project

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

Video Link: https://drive.google.com/file/d/1y-B4cZ79cnsKhnOI0Kd0oSkN9-PWLj7R/view?usp=share_link

Value Numbering is an optimisation technique used to eliminate redundant computations. This technique is useful in eliminating duplicate computations and improve the performance of a program.

Value Numbering is a static program analysis technique for optimizing compilers. It is used to eliminate redundant computations and common subexpressions. The basic idea is to replace all equivalent expressions with a single expression. It is done by assigning a unique number to each expression and comparing the numbers of the expressions.

The idea is to replace the redundant calculations with a single operation in the program. The major data structures used in this technique are a hash table and an integer. The hash table stores the mapping between the original instructions and the generated instructions. The integer is used to store the value of the generated instructions.

Value numbering is used to detect expressions which can be split into components and then the individual components can be consolidated. The major data structure used by the value numbering algorithm is a hash table. This hash table is used to store the expression and their assigned value numbers. The key of the hash table is the expression and the value is the value number assigned to it. The value numbers are assigned incrementally as new expressions are encountered. The implementation of the value numbering algorithm is done by traversing the program and analyzing each instruction.

The algorithm works as follows:

1. Traverse through the instructions in the program and identify the instructions that can be replaced using value numbering.
2. For every identified instruction, check if it is already present in the hash table.
3. If the instruction is present in the hash table, then replace the redundant instruction with the generated instruction.

4. If the instruction is not present in the hash table, then generate a new instruction and add it to the hash table. The implementation of this algorithm is as follows:

• First, the function “addOrFind” is used to search the hash table for the instruction and return the corresponding value if it is found. If the instruction is not found, then it is added to the hash table and a new value is assigned to it.

• Next, for every instruction in the program, the visitor function is used to traverse through the instructions.

• For each instruction, the visitor function checks if it is a load or store instruction. If it is a load instruction, then the operand is added to the hash table and the corresponding value is assigned to it. If it is a store instruction, then the value of the operand is updated in the hash table. For a load instruction, the value of the operand is looked up in the hash table and assigned the corresponding value number. If the operand doesn't exist in the hash table, a new value number is assigned and added to the hash table. For a store instruction, the value of the first operand is looked up in the hash table and the corresponding value number is assigned to the second operand. If the second operand doesn't exist in the hash table, a new value number is assigned and added to the hash table.

• For binary operations, the visitor function checks the operation type and generates a new instruction with the corresponding operands and operation. The generated instruction is added to the hash table and the corresponding value is assigned to it. Finally, the program is optimized by replacing the redundant instructions with the generated instructions. For a binary instruction (add, sub, mul, div), the values of the two operands are looked up in the hash table and a new value number is assigned to the result of the operation. This new value number is then added to the hash table. If the same operation is encountered again, the value number from the hash table is used instead of assigning a new one. The value numbering algorithm is a simple but powerful optimization technique which can be used to reduce the amount of computation required by a program. It can be used in conjunction with other optimization techniques to further improve the performance of the program.

```
#include
"llvm/ADT/ArrayRef.h"
                        #include "llvm/ADT/STLExtras.h"
                        #include "llvm/ADT/SmallString.h"
                        #include "llvm/ADT/SmallVector.h"
                        #include "llvm/ADT/StringRef.h"
                        #include "llvm/Support/FormatCommon.h"
                        #include "llvm/Support/FormatProviders.h"
                        #include "llvm/Support/FormatVariadicDetails.h"
                        #include "llvm/Support/raw_ostream.h"
                        #include "llvm/IR/LegacyPassManager.h"
                        #include "llvm/Passes/PassPlugin.h"
                        #include "llvm/Passes/PassBuilder.h"
```

```
#include "llvm/IR/PassManager.h"
#include <string>
#include <sstream>
#include <map>
#include "llvm/Support/ScopedPrinter.h"
#include "llvm/Support/raw_ostream.h"

using namespace std;

using namespace llvm;

namespace {



       int i=1;
       map<string,int> hashTable;

       int addOrFind(string op){
        auto search = hashTable.find(op);
        auto retVN = i;
        if (search != hashTable.end()){
            retVN = search->second;
        }
        else{
            hashTable.insert(pair<std::string,int>(op,i++));
        }
        return retVN;
    }
// This is were the changes needs to be made
// 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";
           errs() << "ValueNumbering:" << F.getName() << "\n";
           // Comment this line
        //if (F.getName() != func_name) return;




        for (auto& basic_block : F)
        {

            for (auto& inst : basic_block)
            {
                if(inst.getOpcode() == Instruction::Load){
                      string str;
llvm::raw_string_ostream(str)<<inst;

                      string newinst(50,' ');
                      newinst=str;
                      errs() << newinst;
                      string operand1=inst.getOperand(0)-
>getName().str();
                      Value *v = &inst;
                      stringstream ss;
                      ss << v;
                      string value = ss.str();

                      //find, if found get the value, else add it
like i and then return that



                      int op1value=0;
                      op1value=addOrFind(operand1);

                      //overwrite value with operand1's value

       hashTable.insert(pair<string,int>(value,op1value));
                      errs() <<"\t" << op1value << " = " <<
op1value <<"\n";
                      //errs() <<"\t" << operand1 << " = " << value
<<"\n";



                }
                if(inst.getOpcode() == Instruction::Store){
                      string str;
llvm::raw_string_ostream(str)<<inst;

                      string newinst(50,' ');
                      newinst=str;
                      errs() << newinst;
                      string operand1=inst.getOperand(0)-
>getName().str();
              //      Value *v = &inst;
              //      stringstream ss;
              //      ss << v;
              //      string operand1 = ss.str();
                      int j = 1;
                      if(operand1.empty()){

                      auto* ptr = dyn_cast<User>(&inst);
                      for (auto it = ptr->op_begin(); it != ptr-
>op_end(); ++it) {
                             if (j==1){
                      Value *v = dyn_cast<Value>(it);
                      stringstream ss;
                      ss << v;
                      operand1 = ss.str();
                                    j++;
                        }}
                      }

                      string operand2=inst.getOperand(1)-
>getName().str();

                      //operand1 addorfind and get the value
                      int op1value=0;
                      op1value=addOrFind(operand1);
                      //insert the operand2 using the operand1's
value



                      auto search = hashTable.find(operand2);
                      if (search != hashTable.end()){
                      search->second=op1value;
                      }
                      else{

       hashTable.insert(pair<string,int>(operand2,op1value));
                      }
                      errs() <<"\t" << op1value << " = " <<
op1value <<"\n";
                      //errs() <<"\t" << operand1 << " = " <<
operand2 <<"\n";
                }
                if (inst.isBinaryOp())
                {
                      string operand1,operand2,operation,redundant;

                      if(inst.getOpcode() == Instruction::Add){
                        operation = "add";
                    }
                    if(inst.getOpcode() == Instruction::Sub){
                        operation = "sub";
                    }
                    if(inst.getOpcode() == Instruction::Mul){
                        operation = "mul";
                    }
                    if(inst.getOpcode() == Instruction::UDiv){
                        operation = "div";
                    }
               auto* ptr = dyn_cast<User>(&inst);
                auto* val = dyn_cast<Value>(&inst);
               string operand3 = val->getName().str();

               //Value *v = &inst;
               //     stringstream ss;
               //     ss << v;
               //     string operand3 = ss.str();
               int j = 1;

                for (auto it = ptr->op_begin(); it != ptr-
>op_end(); ++it) {
                     if (j==1){
                      Value *v = dyn_cast<Value>(it);
                      stringstream ss;
                      ss << v;
                      operand1 = ss.str();
                        }
                      if (j==2){
                          Value *v = dyn_cast<Value>(it);
                      stringstream ss;
                      ss << v;
                      operand2 = ss.str();
                        }
                        j++;
                    }



               int op1value=0;
               op1value=addOrFind(operand1);
               int op2value=0;
               op2value=addOrFind(operand2);
               int binopvalue=0;
               string
hashkey=to_string(op1value)+operation+to_string(op2value);
               auto search = hashTable.find(hashkey);
               if (search != hashTable.end()){
                redundant=" (redundant) ";
               }
               else{
                redundant=" ";
               }
               binopvalue=addOrFind(hashkey);

       hashTable.insert(pair<string,int>(operand3,binopvalue));



               string str;
llvm::raw_string_ostream(str)<<inst;

                      string newinst(50,' ');
                      newinst=str;
                      errs() << newinst;
               errs() <<"\t" <<"\t"<< binopvalue << " = " <<
op1value << " "+operation+" " <<op2value<< redundant <<"\n";
               //errs() <<"\t" << operand3 << " = " << operand1  <<
operand2 <<"\n";
                } // end if

            } // end for inst
        } // end for block

}
```

```
// 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;
        });
    }};
}
```