DETECTION OF REDUNDANT EXPRESSIONS

A THESIS
Submitted by

ANOOP CHAUDHARY

In partial fulfilment for the award of the degree of

**MASTER OF TECHNOLOGY
IN
COMPUTER SCIENCE AND ENGINEERING**

**Under the guidance of
DR. VINEETH PALERI**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
NATIONAL INSTITUTE OF TECHNOLOGY CALICUT
NIT CAMPUS PO, CALICUT
KERALA, INDIA 673601**

May 14, 2017

ACKNOWLEDGEMENTS

Foremost, I would like to thank my guide Prof. Vineeth Paleri for the continuous support, motivation and immense knowledge and for making me think how to solve complex problems and guiding me in the right direction whenever he thought I need it.

I would like to express my sincere gratitude to the evaluation panel members: Dr. Saleena N, Dr. Sudeep K.S., Dr. Vasudevan N. and Ms. Suvya P. for their advices, insightful comments, and questions that helped me to improve my thought process and project.

Finally, I would like to express my gratitude to my parents and to my friends for providing me support and continuous encouragement.

Anoop Chaudhary

DECLARATION

"I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgment has been made in the text".

Place:	**Signature :**
Date:	**Name :**
	Reg.No:

CERTIFICATE

*This is to certify that the thesis entitled: "***DETECTION OF REDUNDANT EXPRESSIONS***" submitted by Sri/Smt/Ms* **ANOOP CHAUDHARY** *to National Institute of Technology Calicut towards partial fulfillment of the requirements for the award of Degree of Master of Technology in Computer Science and Engineering is a bonafide record of the work carried out by him/her under my/our supervision and guidance.*

Signed by Thesis Supervisor(s) with name(s) and date

Place:
Date:

Signature of Head of the Department

Office Seal

Contents

Abstract

Detection of redundant computations helps in removal of redundant computations. Global Value Numbering (GVN) is one of the method for detecting equivalence among expressions in a program.Considering this I have implemented *A Simple Algorithm for Global Value Numbering* in the LLVM framework in order to detect redundant computations.

Tables

Figures

Chapter 1

Problem Definition

To implement *A Simple Algorithm for Global Value Numbering*[6] in LLVM framework in order to detect redundant computations.

Chapter 2

Introduction

A computation is redundant if it has been already computed before the point under consideration. Redundant computations can be removed and this can improve the running time of program.But it needs efficient detection strategies.Most of the redundant computations can be identified at the compile time.Mostly redundancy elimination algorithm are of two kinds: lexical algorithms and value number algorithms. Common sub expression elimination can detect redundancy among computations of lexically identical expressions.Two expressions are lexically identical if they apply same operators to the same operands.This technique does not detect some redundancies that can be determined at compile time.On the other hand value numbering algorithms can identify redundancy among expressions that are lexically different. Global value numbering (GVN) can detect redundant computations that common sub expression elimination (CSE) does not.

This project is aimed at implementing *A Simple Algorithm for Global Value Numbering* in LLVM framework. Working for the project started with understanding of the LLVM framework and reading the algorithm [6]. Then implementation of passes: *Simple Constant Propagation Pass* and *Common Subexpression Detection Pass* is done in order to familiarize with LLVM frame work. Finally I have implemented *A Simple Algorithm for Global Value Numbering* [6] in LLVM framework.

Chapter 3

Literature Survey

A Simple Algorithm for Global Value Numbering[6] paper is referred for implementation of algorithm[6] in LLVM framework. Compilers: Principles, Techniques and Tools [7] is used to understand basics of compiler design. Basics of LLVM framework is learnt from [4]. Installation is done using paper [1]. Writing *A Sample pass* is done using paper [3]. To understand about LLVM intermediate representation paper [5] is referred. For implementing algorithm[6] in LLVM framework LLVM Programmer's manual and LLVM Classes is used . To solve questions related to LLVM framework , a interactive platform *llvm-dev Google Group* is used.

Chapter 4

LLVM

4.1 What is LLVM?

LLVM is a compiler framework with a collection of tools that helps in compiler building. It provides modular and reusable compiler and tool-chain technologies.[4] Certain built-in libraries of the LLVM framework can be used to do specific tasks.
LLVM began as a research project at the University of Illinois under the direction of Vikram Adve and Chris Lattner, with the goal of providing a modern, SSA-based compilation strategy capable of supporting both static and dynamic compilation.[4]
LLVM is written in C++ and is designed for compile-time, link-time, run-time optimization . Originally implemented for C and C++.Languages with compilers that use LLVM are ActionScript, Ada, C#, Common Lisp, Crystal, D, Delphi, Fortran Haskell, Java bytecode etc.

4.2 LLVM Architecture

LLVM framework has the three phase design. Its major components are the front-end, the optimizer and the back end.[4]
Front end is a translator for any high level language to the LLVM intermediate representation.It performs parsing, validating and diagnosing errors in the input

code, then translating the parsed code into LLVM intermediate representation, this intermediate representation is optionally fed through a series of analysis and optimization passes which improve the code (done by optimizer), and then code generator produces native machine code.
Modular design facilitates supporting multiple languages and multiple machines just by changing the front and the back ends. Clang is a front end that compiles C language code to the LLVM Intermediate representation.

4.3 LLVM Intermediate representation

LLVM intermediate representation is a Static Single Assignment (SSA) based representation that provides type safety, low-level operations, flexibility, and the capability of representing all high-level languages cleanly.[5] It is the common code representation used throughout all phases of the LLVM compilation strategy. The LLVM code representation mainly used in three different forms text, binary, and in-memory forms.The three different forms of LLVM IR are all equivalent. LLVM intermediate representation is a low-level RISC-like virtual instruction set. These instructions are in three address form, It supports linear sequences of simple instructions like add, subtract, compare, and branch.LLVM IR uses an infinite set of temporaries named with a % character.[4]
LLVM intermediate representation is strongly typed with a simple type system (e.g.i32 is a 32-bit integer) and some details of the machine are abstracted away. For example, the calling convention is abstracted through *call* and *ret* instructions and explicit arguments.

4.4 Installing LLVM and Clang [1]

Given below are step-by-step instructions to build LLVM and Clang from source on Linux system.

(1) Installing cmake 3.4.3

   (a) *sudo apt-get install build-essential*

   (b) *wget http://www.cmake.org/files/v3.4/cmake-3.4.3.tar.gz*

   (c) *tar xf cmake-3.4.3.tar.gz*

   (d) *cd cmake-3.4.3*

   (e) *./configure*

   (f) *make*

   (g) *sudo make install*

(2) Get source code

   (a) *mkdir LLVM*

   (b) *cd LLVM*

   (c) *svn co http://llvm.org/svn/llvm-project/llvm/trunk llvm* (Get LLVM source code)

   (d) *cd llvm/tools*

   (e) *svn co http://llvm.org/svn/llvm-project/cfe/trunk clang* (Get clang source code)

   (f) *cd llvm/projects*

   (g) *svn co http://llvm.org/svn/llvm-project/compiler-rt/trunk compiler-rt* (Get compiler-rt source code)

(3) Build LLVM and Clang

   (a) *cd LLVM*

   (b) *mkdir build*

   (c) *cd build*

(d) *cmake -G "Unix Makefiles" ../llvm*

(e) *make*

(f) *sudo make install*

(This builds both LLVM and Clang for debug mode.For subsequent Clang development, you can just run make clang.)

4.5 Writing a new pass (Sample Hello pass) [3]

(1) *mkdir LLVM/llvm/lib/Transforms/Hello*

(2) *cd LLVM/llvm/lib/Transforms/Hello*

(3) Write sample pass in *Hello.cpp* (Refer *http://llvm.org/docs/WritingAnLLVMPass.html*)

(4) Write a Makefile (Refer *http://llvm.org/docs/WritingAnLLVMPass.html*)

(5) *cd ..*

(6) Edit Makefile in *Transforms* directory Add *Hello* in *add_subdirectory* list

(7) *cd LLVM/build*

(8) *make*

Successful compilation can be verified by *Hello.so* shared library in *LLVM/build/lib*

Chapter 5

Writing a simple Constant propagation Pass

To familiarize with LLVM framework *A simple Constant Propagation Pass* is written. The goal of constant propagation is to find values that are constant in a program and to propagate these constant values through the program. Constants assigned to a variable can be propagated through the flow graph and substituted at the use of the variable. In a program having a statement like: $z = x + y$ *(in three address form)* here x can be replaced by constant value by checking definition of x reaching at this point.In case of linear control flow we can easily replace x with constant value if definition of x reaching at this point have constant value.

Let $x = 5$ is the definition of x reaching at given program point we can see on it's R.H.S has constant value so we can replace use of x in $z = x + y$ by R.H.S of x=*5* i.e *5*. For non linear control flow if given program point have multiple control flow paths incoming then we can replace x by constant only if all the definitions of x reaching at this point have same constant value.

Example 1

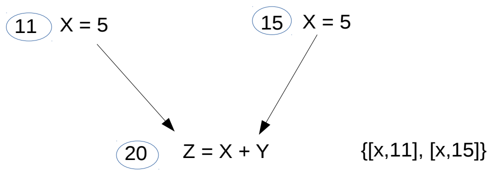


Figure 5.1: Example 1

In Figure 5.1 we can replace x by *5* at program point 20. As *x's* definitions reaching at point 20 are {*[x,11], [x,15]*} have same constant value on it's R.H.S.

Example 2

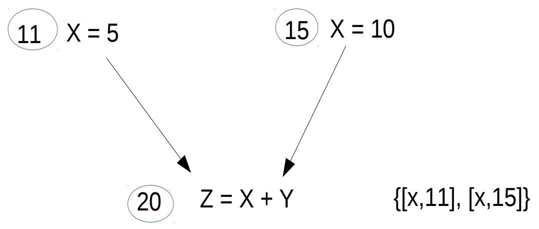


Figure 5.2: Example 2

In Figure 5.2 we can not replace x by constant value at program point 20. As *x's* definitions reaching at point 20 are {*[x,11], [x,15]*} have different constant value on it's R.H.S.

Consider the example given below to do constant propagation on LLVM IR.

C program

int main(){

$\quad int\ a, b, c;$

$\quad a = 1;$

$\quad b = 2;$

$\quad c = a + b;$

}

Corresponding LLVM IR in text form

```
define i32 @main() #0 {
entry:
%a = alloca i32, align 4
%b = alloca i32, align 4
%c = alloca i32, align 4
store i32 1, i32* %a, align 4
store i32 2, i32* %b, align 4
%0 = load i32, i32* %a, align 4
%1 = load i32, i32* %b, align 4
%add = add nsw i32 %0, %1
store i32 %add, i32* %c, align 4
ret i32 0
}
```

To replace *a* with constant in statement $c = a + b$ we have to check all the definitions of *a* reaching at this point are constant and all of them have same value . Definition of *a* is reaching from statement *a = 1* and it's corresponding LLVM IR is *store i32 1, i32* %a, align 4* .To check definitions have constant value we need to check R.H.S of statement *a=1* and this is similar to checking *operand(0)* of *store i32 1, i32* %a, align 4*. Here *operand(0)* has constant value *1*. So we can replace *a* with *1* in statement $c = a + b$. But LLVM IR corresponding to statement $c = a + b$ is *%add = add nsw i32 %0, %1* where *%0* and *%1* are *Load* Instruction. *%0* is from Instruction *%0 = load i32, i32* %a, align 4* . To check temporaries *%0* and *%1* has constant value, we need to check value stored at address specified by operand of *load* instruction. For *%0* address is of *operand[0]* (i.e. *i32 * %a*) from *load* instruction. But at specified address ** %a* of load Instruction, constant value is stored by Instruction *store i32 1, i32* %a, align 4*. So we need to check *load* along with *store* instruction to find constant value. Above example have linear control flow so we can replace *%0* by *1*. In case of non-linear control flow we need to check definitions of *%0* reaching from all paths have same constant value.

In implementation I have considered only linear control flow.

As we can see a register contains constant value can be checked by checking *load* and corresponding *store* instruction. A map (data structure) $DenseMap < Value*, Instruction* > map$ is used in LLVM constant propagation Pass to get correct *store* Instruction for *load* Instruction.This data structure is available in $llvm/ADT/DenseMap.h$.This map keeps pair of pointer operand of *store* Instruction and reference to *store* Instruction. This map can be built by iterating over all store Instructions in LLVM IR.Later this map is used to get constant value from store Instruction. After making map we will iterate over all load instructions and we will check stored value at address specified by load instruction. If it's a constant we will replace all of it's uses with constant value. In above example *%add = add nsw i32 %0, %1*, where *%0* corresponds to *%0 = load i32, i32* %a, align 4* and it specifies address *i32 * %a*, at this address *1* is stored by *store i32 1, i32* %a, align 4* . So we will replace all uses of *%0* with *1* by checking into map. For replacement of all uses of load instruction with constant value operand of store Instruction a function from *Value* class "$void\ replaceAllUsesWith(Value*)$" is used in Pass.

LLVM IR in text form after running Constant Propagation Pass

```
define i32 @main() #0 {
entry:
%a = alloca i32, align 4
%b = alloca i32, align 4
%c = alloca i32, align 4
store i32 1, i32* %a, align 4
store i32 2, i32* %b, align 4
%0 = load i32, i32* %a, align 4
%1 = load i32, i32* %b, align 4
%add = add nsw i32 1, 2
store i32 %add, i32* %c, align 4
ret i32 0
}
```

Here constant value stored in *%a* and *%b* gets propagated to *add* Instruction after running pass.

Chapter 6

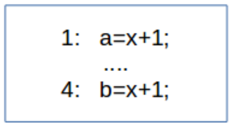
Writing Common Subexpression Detection Pass

Common subexpression refers to an expression which is previously computed and the values of the variables in expression have not changed since the previous computation. Common Subexpression detection is used to detect re-computation of such expressions by using previously computed expressions.In Common Subexpression detection we search for lexically identical expressions. Common Subexpression Detection is performed based on the Available Expression Analysis(data flow analysis).Detection of Common Subexpression helps to perform Common Subexpression Elimination. In order to implement Common Subexpression detection in LLVM framework first I have understood Available Expression Analysis and Detection of Common Subexpressions. Then I have implemented Available Expression Analysis pass in LLVM and using this pass I have implemented Common Subexpression Detection in LLVM framework. Firstly I have written about Common Subexpression Detection at the conceptual level and then discussed about implementation level.

6.1 Common Subexpression Detection

In CSE we replace the computation of available expression by a previously computed value. In order to do this we need to detect such common subexpressions. An occurrence of an expression e is called a common subexpression if e was previously computed and the values of the variables in e have not changed since the previous computation. Re-computation of e can be avoided if we can use its previously computed value[7].

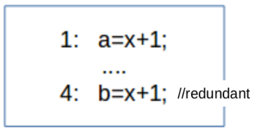
Example 1:



```
1:  a=x+1;
    ....
4:  b=x+1;
```

Figure 6.1: CFG for Example 1

In above example *x+1* is computed twice and value of variable x hasn't changed in between statement *1* and *4*. Hence *x+1* is a common subexpression . CFG after Detection of Common Subexpression is given below .



```
1:  a=x+1;
    ....
4:  b=x+1;  //redundant
```

Figure 6.2: CFG after Detection of Common Subexpressions

To perform such common subexpression detection we need Available Expression Analysis.

6.2 Available Expression Analysis

An expression x+y is available at a point p if every path from the entry node to p evaluates x+y, and after the last such evaluation prior to reaching p, there are no subsequent assignment to x or y[7].

Available expression analysis computes set of expressions available at input and output point of a node n.

6.2.1 Notation

Set of expressions available at input point of a node are represented by *in[n]*.

Set of expressions available at output point of a node are represented by *out[n]*.

kill : A block kills expression $x+y$ if it assigns x or y and doesn't compute subsequently $x+y$.

gen: A block generates an expression if it evaluates $x+y$ and doesn't subsequently define x or y.

Set is the data structure used here enables various set operations to be used by the analysis. Data flow equations for available expressions given below.

$$out[b] = gen[b] \cup (in[b] - kill[b])$$
$$in[b] = \cap_{p\ a\ predecessor\ of\ b}\ out[p]$$

6.3 Detection of Common Subexpression

- Compute set of Available expressions at input and output point of each Basic Block.
- For each statement *S: $a = b$ op c* such that b *op* c is available at entry of basic block of S and neither b or c are redefined prior to S do the next step.

  (a) Mark statement S have redundant
  common subexpression.

Example 2:

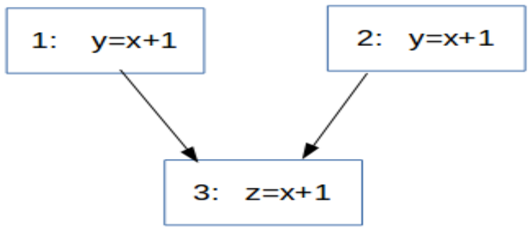


Figure 6.3: CFG

CFG after performing Available expression analysis on above example

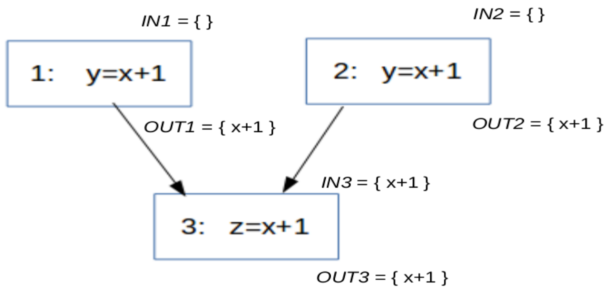


Figure 6.4: CFG after Available expression Analysis

At statement *3: z = x + 1* , expression *x + 1* is available at entry of basic block of *3* and *x* is not redefined prior to statement *3* so we can mark statement *3* have redundant common subexpression. CFG after detection of common subexpressions.

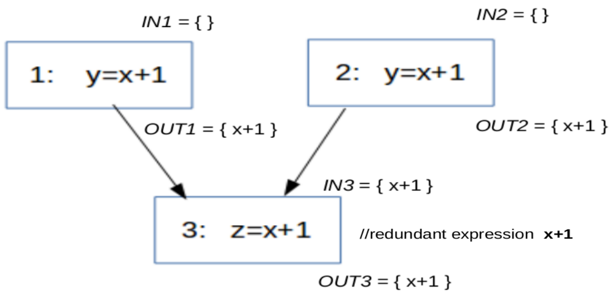


Figure 6.5: CFG after Detection of Common Subexpression

6.4 Implementation details of Common Subexpression detection Pass in LLVM framework

6.4.1 Data Structure (conceptual)

Available expressions at input and output point of each Basic Block is represented by Set Data Structure (Set of expressions) . In set of expressions each expression is represented by a structure with three members: the first holds the *operator*, the next two holds the *operand1* and *operand2* respectively.

6.4.2 Data Structure (Implementation in LLVM)

In implementation an expression is represented by a pointer to instruction and set of expressions is represented by set of pointers to instruction. Instead of keeping expression using structure (with three members) I have kept instruction pointers. Using this pointer to instruction we can access its operator and operands easily.
For representing set in LLVM, vector (dynamic array) is used.
Set of expressions at Input point of each Basic Block is represented by *IN*:

SmallVector<Instruction> IN*;

Similarly set of expressions at Output point of each Basic Block is represented by *OUT*: *SmallVector<Instruction*> OUT*;

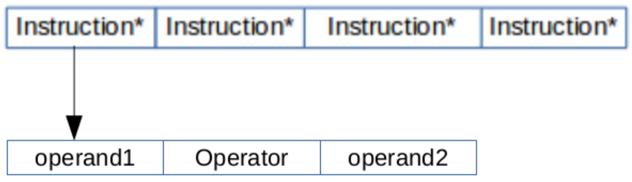


Figure 6.6: Representation of set of expressions

6.5 Algorithm and Functions used in LLVM Pass

To compute set of Available expressions at input and output point of each Basic Block I have used *iterative algorithm to compute available expressions* from *Compilers: Principles, Techniques and Tools* [7].

6.5.1 Transfer function

Conceptually, the transfer function of a statement s, takes the data flow value before the statement and produces a new data flow value after the statement. Transfer function of basic block can be derived by composing transfer function of the statements in Basic Block[7].

$$out[b] = gen[b] \cup (in[b] - kill[b])$$

In implementation available expressions at output point of Basic Blocks are computed by function named **transferFunction(BasicBlock &BB)** in LLVM, which takes Basic Block reference as input. Using Basic Block reference we can access all it's instructions (statements) one by one. Written *transfer* function is different from given in book [7], since I have used LLVM IR in SSA form after running *mem2reg* (so no need to perform kill operation since each variable is assigned once only). I have used SSA form to ease the complexity of computation of available expressions. Firstly, it adds all expressions from **IN** of Basic Block to **tempOUT** (dynamic array of structures) of Basic Block and then it checks each expression from Basic Block whether it's in **IN** set or not (using instruction pointer and **compareInst** method). If it is not found in **IN** set it adds that expression (i.e. Instruction pointer) to **tempOUT** set.Finally at the end of basic block it assigns **tempOUT** to **OUT** of Basic Block.

Comparison of expressions : Conceptually comparison of two expressions is done (lexically) by comparing their operators and left operand and right operand respectively In implementation comparison of expressions is done by function **compareInst(Instruction *i1, Instruction *i2)**. Which takes pointers to two instructions and compares operators using **getOpcode()** method and comparison of corresponding left and right operand done by using **getOperand(0)** and **getOperand(1)** method.

6.5.2 Meet operator

Conceptually Available expressions at input point of Basic Block is computed by *meet* operator. It takes intersection of set of available expressions at output point of all of predecessor Basic Blocks of current Basic Block.Input to meet operator is information (Set of available expressions) available at output point of all predecessor Basic Blocks.

$$in[b] = \cap_{p \text{ a predecessor of } b}\ out[p]$$

In implementation this meet operator is implemented by function **meet(BasicBlock &BB)**, which takes current Basic Block reference as input. Using current Basic Block reference we can access all of it's predecessors and set of available expressions at output point of these predecessor Basic Blocks in LLVM.

6.6 LLVM pass for Common Subexpression detection

Written pass works on LLVM IR after converting it into SSA form using *mem2reg* pass.I have considered *ADD* instruction for writing pass. Working of pass is illustrated by taking an example below.

Example (C++ Program)

```
int main(){
    int x = 0, y;
    if(x > 0)
        y = x+1;
    else
        y = x+1;
    y = x+1;
}
```

CFG(for above Program)

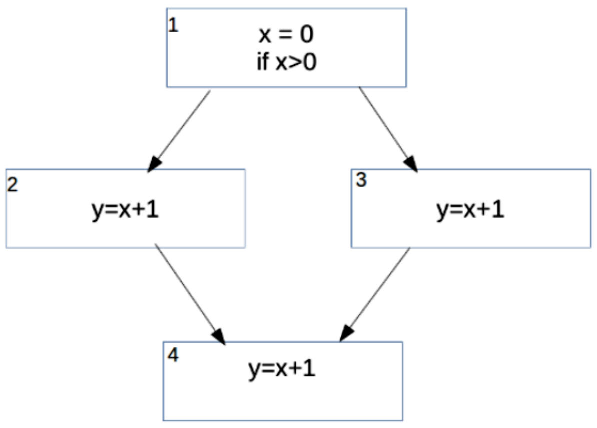


Figure 6.7: CFG of program

In both predecessor Basic Blocks of Basic Block *4* has computation of *x+1* so expression *x+1* is available at *IN* point of the Basic Block *4*. So the computation of expression *x+1* in Basic Block *4* is redundant, as there is no reassignment to variable x prior to reaching y = *x+1* in Basic Block *4*.

CFG(SSA form,not implemented)

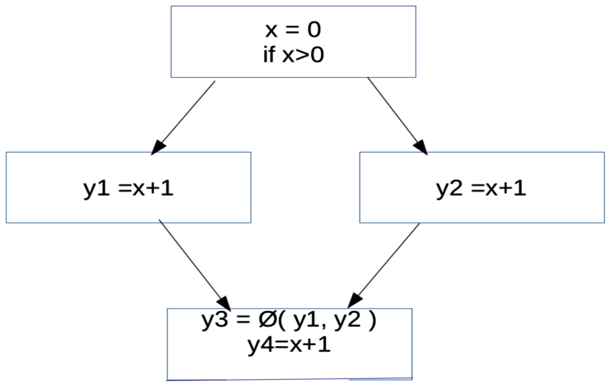


Figure 6.8: CFG of program (SSA form)

Here we will explain how to detect the redundant expressions using LLVM pass.Written pass works on LLVM IR in SSA form . In first step we obtained LLVM IR in text form corresponding to *C++* program given above. Then we use *mem2reg* pass to obtain LLVM IR in SSA form.Then we run our *newCSE* pass on LLVM IR obtained from running *mem2reg pass.*

6.7 Steps to run *newCSE* pass

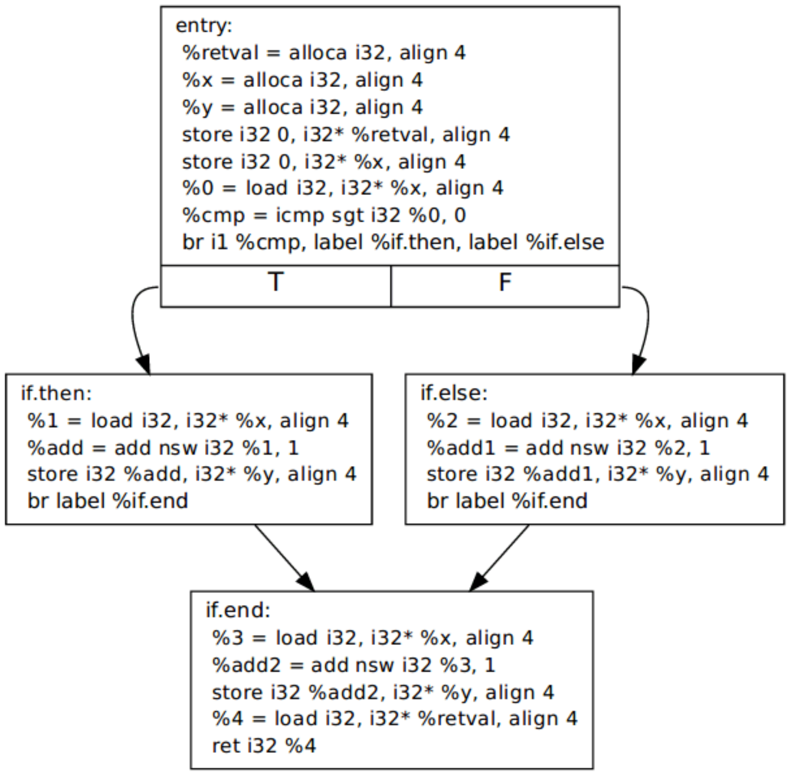
6.7.1 Step 1:

Generation of LLVM IR in text form from *C++* code by command:-

clang -emit-llvm -g -S hello.cpp -o hello.ll

#hello.cpp contains C++ code given in example

LLVM IR(for CFG original, result of Step 1)



```
entry:
 %retval = alloca i32, align 4
 %x = alloca i32, align 4
 %y = alloca i32, align 4
 store i32 0, i32* %retval, align 4
 store i32 0, i32* %x, align 4
 %0 = load i32, i32* %x, align 4
 %cmp = icmp sgt i32 %0, 0
 br i1 %cmp, label %if.then, label %if.else
```

```
if.then:
 %1 = load i32, i32* %x, align 4
 %add = add nsw i32 %1, 1
 store i32 %add, i32* %y, align 4
 br label %if.end
```

```
if.else:
 %2 = load i32, i32* %x, align 4
 %add1 = add nsw i32 %2, 1
 store i32 %add1, i32* %y, align 4
 br label %if.end
```

```
if.end:
 %3 = load i32, i32* %x, align 4
 %add2 = add nsw i32 %3, 1
 store i32 %add2, i32* %y, align 4
 %4 = load i32, i32* %retval, align 4
 ret i32 %4
```

Figure 6.9: CFG of LLVM IR obtained by step1

LLVM IR obtained by *step 1* doesn't have memory objects in SSA form although virtual registers are in SSA form, to transform it into SSA we will run *mem2reg* pass first.

6.7.2 Step 2:

Converted LLVM IR in SSA form using *mem2reg* pass by using command.

opt -mem2reg -S < hello.ll > ./hello1.ll

#hello1.ll contains LLVM IR in SSA form

LLVM IR (after step 2)

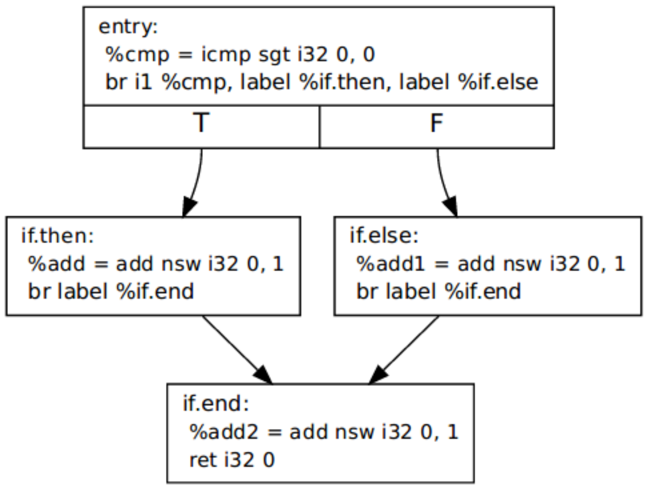


Figure 6.10: CFG of LLVM IR after *mem2reg* pass

Since *add nsw i32 0, 1* is available at input point of Basic Block *if.end* and *if.end* Basic Block have computation of expression *add nsw i32 0, 1* (from instruction *%add2 = add nsw i32 0, 1)* which is redundant. This re-computation of expression is detected by running *newCSE* pass on *hello1.ll*.

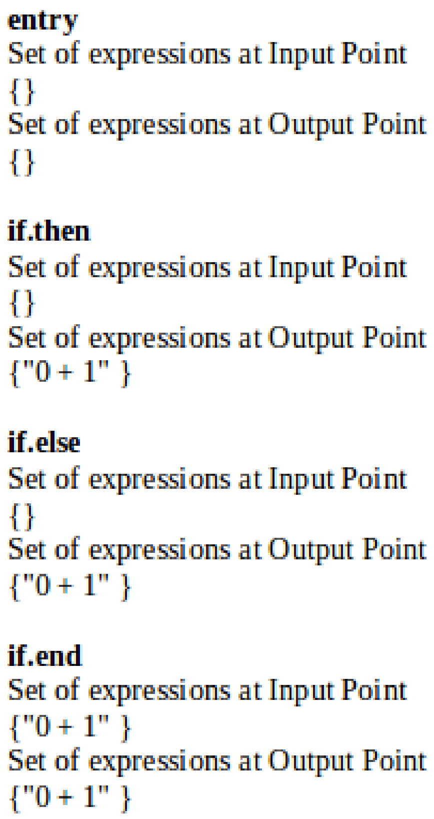
6.7.3 Step 3:

Obtain Set of Available expressions at input and output point of each Basic Block and redundant common subexpressions in program by running *newCSE* pass by using command.

opt -load /home/anoop/LLVM/ build/lib/LLVMHello.so -newCSE -S <hello1.ll

6.8 Result

6.8.1 Available expressions Analysis Result



```
entry
Set of expressions at Input Point
{}
Set of expressions at Output Point
{}

if.then
Set of expressions at Input Point
{}
Set of expressions at Output Point
{"0 + 1" }

if.else
Set of expressions at Input Point
{}
Set of expressions at Output Point
{"0 + 1" }

if.end
Set of expressions at Input Point
{"0 + 1" }
Set of expressions at Output Point
{"0 + 1" }
```

6.8.2 Redundant expressions in LLVM IR(SSA)

Line no 7 in c++ program.

Corresponding expression in LLVM IR(SSA): ”0 + 1”.

6.9 Commands to run *newCSE* Pass

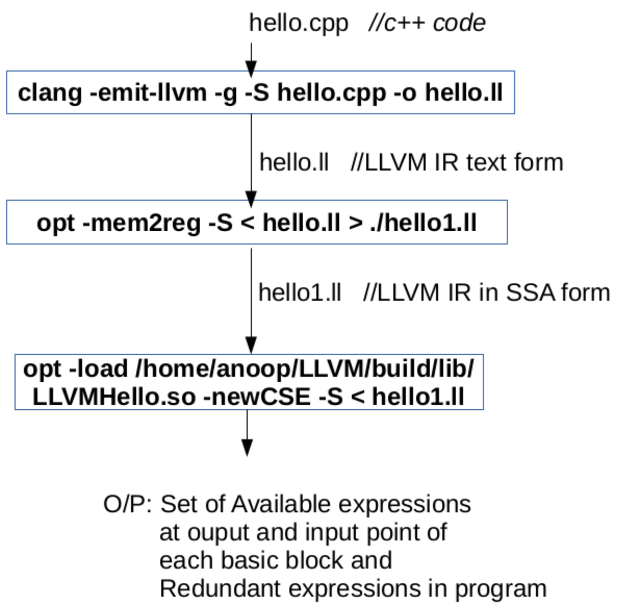

hello.cpp //c++ code
clang -emit-llvm -g -S hello.cpp -o hello.ll
hello.ll //LLVM IR text form
opt -mem2reg -S < hello.ll > ./hello1.ll
hello1.ll //LLVM IR in SSA form
opt -load /home/anoop/LLVM/build/lib/LLVMHello.so -newCSE -S < hello1.ll
O/P: Set of Available expressions at ouput and input point of each basic block and Redundant expressions in program

Figure 6.11: Commands to run *newCSE* pass

Chapter 7

Implementation of ***A Simple Algorithm For Global Value Numbering*** in LLVM framework

7.1 Understanding Algorithm

To implement *A Simple Algorithm for Global Value Numbering* [6] in LLVM framework I have thoroughly studied the paper. In order to detect redundant computations in a program Global Value Numbering(GVN) can be used. The algorithm uses the concept of value expression (expression with value number as operand)- an abstraction of a set of expressions enabling a representation of the equivalence information which is compact and simple to manipulate[6]. It extends the idea of local value numbering to detect redundant computations globally. It assigns a number to each expression in such a way that equivalent expressions get same value number. Input to the algorithm is flow graph. Node can have one assignment statement in three-address code. Maintains expression pools at input and output points of every node in graph.Uses EIN_n and $EOUT_n$ to denote expression-pools at IN_n and OUT_n respectively[6] for a node n. $EOUT_n$ is computed by means of transfer function for a node containing the assignment statement $x = e$. Node point where multiple control flow path merge (confluence point) the equivalence information (expression pools) is computed by using confluence operator. Later this equivalence information can be used to detect redundant computations.

7.2 Data Structure (Conceptual)

7.2.1 Expression Pool

Expression pools at input and output point of each node are represented by Set data structure (Set of equivalence classes).

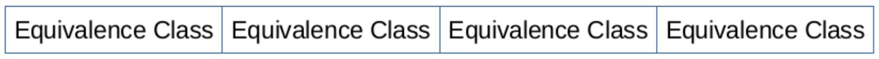


Figure 7.1: Set of Equivalence Classes as Expression Pool

7.2.2 Equivalence Class

In Set of equivalence classes each equivalence class is represented by Set data structure (Set of expressions). Each equivalence class has a value number as first element in it .We will divide Set of expressions in categories: constant, variables and expressions of form $x\ op\ y$ (represented by value expression) to store expressions efficiently. To do this will use structure to represent Equivalence class.

Equivalence class structure consists of :

(1) **valueNumber** (e.g. *v1*)

(2) **constant** (e.g. *5*)

(3) **variablesSet** (e.g. x, y)

(4) **valueExpressionStruct** (e.g. *v1 + v2*)

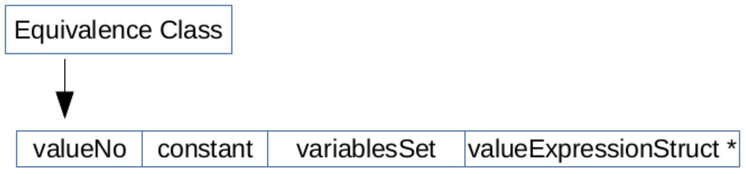


Figure 7.2: Representation of Equivalence Class

7.2.3 Expression

In Set of expressions each expression (is either a constant, a variable, or an expression of form x op y) is represented by a structure with three members: the first holds the *operand1*, the next two holds the *operator* and *operand2* respectively.

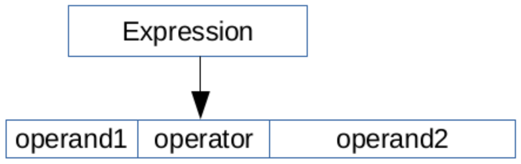


Figure 7.3: Representation of Expression

7.2.4 Value Expression

For expressions of form *x op y* we keep value expression instead of expressions in expression pool. Value expression is obtained by replacing operands with their corresponding value number.

Value expression represented by structure consists of :

(1) value number of *operand1*

(2) *operator*

(3) value number of *operand2*

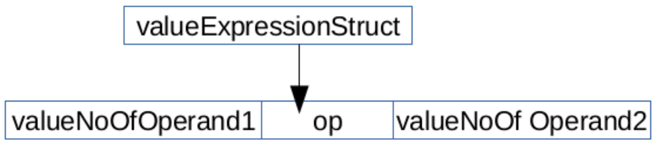


Figure 7.4: Representation of value Expression

7.3 Data Structure (in LLVM)

7.3.1 Value Number

Value number is represented by *integer* number (for v1 we will keep *integer 1*). A variable *GlobalValueNo* is used to assign value number to new equivalence class.

7.3.2 Expression

An expression is either a constant, a variable or an expression of form *x op y*. Constant and variables are represented by reference to *Value* (i.e *Value**, constants and variables are *Value* in LLVM IR). For expression of form *x op y* we will keep its value expression. Its structure **valueExpressionStruct** consists of :

(1) **operand1** of type *int*

(2) **op** of type *char*

(3) **operand2** of type *int*

(e.g. for expression $x + y$ corresponding value expression is $v1 + v2$ stored in form of $1 + 2$).

7.3.3 Equivalence Class

Each equivalence class has a value number and class can have constant and variables and at most one value expression. Representing expressions using structure with three members (operand1, operator ,operand2) is memory inefficient for constants and variables.
Equivalence class is represented by structure **equivalenceClassStruct**. It consists of :

(1) **valueNo** of *integer* type (for *v1*, valueNo is integer *1*)

(2) **constVarSet**: Set of Constant and Variables of type *SetVector< Value* >*

(3) **valueExp** of type *valueExpressionStruct*

7.3.4 Expression Pool

Set of equivalence classes is represented by hash map (pairs keys to values) data structure. In which each pair contains **valueNo** as a key and address of **equivalenceClassStruct** as a value.

Expression Pool′s type in LLVM is:

DenseMap<int,equivalenceClassStruct >*.

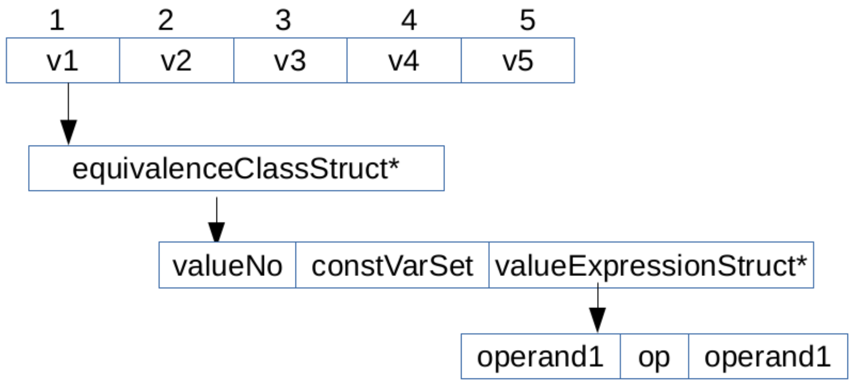


Figure 7.5: Representation of Expression Pool in LLVM

Expression pool at input point of a node(contains a statement i.e. Instruction) is represented by **EIN**:

DenseMap< int, equivalenceClassStruct > EIN*

Expression pool at output point of a node is represented by **EOUT**:

DenseMap<int, equivalenceClassStruct> EOUT*

7.4 Algorithms and Functions used in implementation

7.4.1 Transfer function (conceptual)

Transfer function of a statement s, takes the data flow value before the statement and produces new data flow value after the statement. Transfer function associated with a node n(denoted by f_n) computes $EOUT_n$ using EIN_n .

$$EOUT_n = f_n(EIN_n) \text{ [6]}$$

Transfer function for a node n containing an assignment $x = e$ (x: a program variable, e: an expression) performs fallowing steps:

(1) **Killing all expressions involving variable x**

 (a) Removes x from its class $C_i \in E_i$.

 (b) If C_i is singleton after *Step (a)* (value number v_i only element in it) then remove class C_i and any value expression involving v_i from pool (*deleteSingletons(E)* does this repeatedly till no singleton class remain in pool).

(2) **Generates new equivalence between x and e**

 (a) Finds value expression of e. If e is of the form x *op* y value expression is obtained by replacing operands by its corresponding value number else e is itself considered as value expression.

 (b) If e is already assigned a value number (in EIN_n) then insert x in same equivalence class of e else create new class containing x and e together (if e is of form x *op* y then add value expression of e instead of e itself) and add distinct value number in it, then add this new class to output pool.

7.4.2 Transfer function (in LLVM)

Transfer function in LLVM takes two inputs: Instruction reference and old *EOUT* (*EOUT* of instruction in previous iteration) of instruction as argument. Instead of taking *EIN* as input we took Instruction reference. Using Instruction reference we can access *EIN* of Instruction and we can use it to access operands and operator of corresponding expression too. Old *EOUT* is used to avoid assigning new value number to same equivalence class in every new iteration.
Transfer function in LLVM considers each node with a single instruction (i.e. statement conceptually). Based on instruction it computes the *EOUT* pool. Transfer function handles fallowing instructions:

(1) **Load Instruction**:

Example: *%tmp = load i32, i32∗ %e, align 4.*

Generates new equivalence between *tmp* and *e* using second step written in section *7.4.1.* Killing of expressions involving *tmp* is not done as LLVM IR is in SSA form.

(2) **Store Instruction** (equivalent to assignment statement conceptually):

Example: *store i32 %add, i32* %x, align 4.* First it kills all expressions involving variable *x* (as memory objects are not in SSA form in LLVM IR) then generates new equivalence between *x* and *add* using first and second step written in section *7.4.1* .

(3) **Instruction with binary operator**

I have considered these binary operators in implementation *+, -, *, /, %, &&, ||, XOR.*

Example: *%add = add nsw i32 %tmp, %tmp1.* Generates new equivalence between *add* and *tmp + tmp1* (using second step from section *7.4.1*).

7.4.3 Confluence of expression pools (conceptual)

At confluence point, expression pool contains equivalent expressions common to all incoming pools[6]. Computation of confluence of two expression pools E_i and E_j can be done by taking each pair of classes $C_i \in E_i$ and $C_j \in E_j$, and finding common expressions (explicitly present or implicitly represented by value expression) in C_i and C_j using special intersection ($C_i \sqcap C_j$).

7.4.4 Confluence of expression pools (in LLVM)

In LLVM confluence of two expression pools is performed by function *confluence(E_i,E_j)* takes two expression pools E_i and E_j, returns pool E_k, and does fallowing steps:

(1) Creates a new empty pool E_k.

(2) Takes each pair of classes, $C_i \in E_i$ and $C_j \in E_j$ then computes $C_i \sqcap C_j$. Let $C_k = C_i \sqcap C_j$. If C_k is not empty it adds C_k to E_k .

(3) performs *deleteSigletons(E_k).*

(4) Finally returns E_k.

Special intersection ($C_i \sqcap C_j$)[6] Finding expressions that are common in classes $C_i \in E_i$ and $C_j \in E_j$ can be done by special intersection. To compute $C_i \sqcap C_j$, let class C_k is empty, add elements (value number and expressions) that are common between C_i and C_j to C_k using simple intersection of two classes C_i and C_j. If classes C_i and C_j have different value expressions, let $v_{i1}+v_{i2}$ in C_i and $v_{j1}+v_{j2}$ in C_j be the value expressions then to compute common expressions represented by these two value expressions in class C_i and C_j, we will take intersection of the classes of v_{i1} and v_{j1}, let it results in class with value number v_{k1} and intersection of classes of v_{i2} and v_{j2} results in class with value number v_{k2} . If classes of v_{k1} and v_{k2} are non empty, then pair of value expressions $v_{i1}+v_{i2}$ and $v_{j1}+v_{j2}$ represent a common expression e (value expression of e is $v_{k1}+v_{k2}$) and then we will add value expression of e to C_k. If C_k is not empty and C_k does not have a value number we will assign a new value number to C_k.

7.4.5 Special intersection (In LLVM)

In LLVM special intersection is done by *specialIntersection(C_i,C_j,E_i,E_j)*. Expression pools E_i and E_j are passed as argument in *specialIntersection*, these are used to access classes corresponding to value number. It performs following steps:

(1) Creates new empty class C_k.

(2) If C_i and C_j have same value number it adds same value number of C_i to C_k.

(3) Inserts common variables and constant between C_i and C_j to C_k.

(4) If C_i and C_j have same value expression then it adds same value expression of C_i to C_k.

(5) If C_i and C_j have different value expression $v_{i1}+v_{i2}$ and $v_{j1}+v_{j2}$ respectively and intersection of classes of v_{i1} and v_{j1} results in non empty class with value number v_{k1}, and intersection of classes of v_{i2} and v_{j2} results in non empty class with value number v_{k2}, then it adds value expression $v_{k1}+v_{k2}$ in C_k.

(6) If C_k is not empty and C_k does not have a value number then adds a new value number v_k to C_k.

(7) Finally returns C_k.

7.5 Other implementation details

(1) *int findValueNo(Value∗ x, DenseMap<int, equivalenceClassStruct∗ > & ExpPool)*:

It finds value numbers of x in pool (*ExpPool*).

(2) $int\ findValueNoOfExp(valueExpressionStruct*$ *valExp, DenseMap <* $int, equivalenceClassStruct* >$ & *ExpPool)*:

Finds value number of value expression in pool.

(3) *deleteSingletons(DenseMap<int, equivalenceClassStruct∗ > & ExpPool)*:

Performs deletion of singleton classes.

(4) *bool compareExpPools(DenseMap<int ,equivalenceClassStruct ∗ > & oldEout, DenseMap<int ,equivalenceClassStruct ∗ > & newEout)*:

Compares *newEout* to *oldEout* by checking change in equivalence information.

(5) *map<std::pair<* $int, int >, int >$ *intersectionTable*:

If $C_i \sqcap C_j$ results in class C_k it keeps $(i, j)- > k$ in table. It is used to assign same value number to resultant class of intersection of same pair of classes.

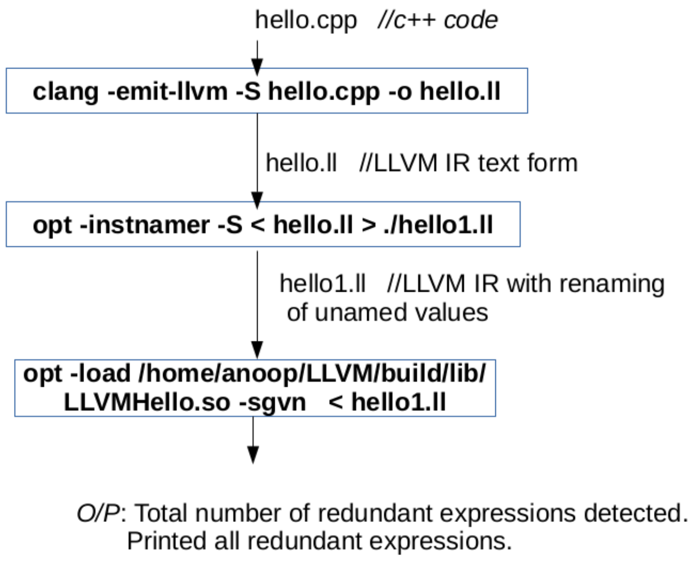
7.6 Results

Implemented pass *sgvn* to detect the number of redundant expressions in SPEC CPU2006. I have considered expressions with binary operator $+, -, *, /, \%$ for redundant expressions detection. Table is give below, lists the result of number of redundant expressions detected in SPEC CPU2006 programs.

Program	**Number of redundant expressions detected**
astar	26
bzip2	54
gcc	161
gromacs	455
h264ref	2016
hmmer	526
lbm	794
mcf	36
povray	307
sjenp	226
soplex	39
sphinx	60

Table 7.1: Result of SPEC CPU2006 benchmark programs

7.7 Commands to run *sgvn* Pass


hello.cpp //c++ code
hello.ll //LLVM IR text form
hello1.ll //LLVM IR with renaming of unamed values

```
clang -emit-llvm -S hello.cpp -o hello.ll
opt -instnamer -S < hello.ll > ./hello1.ll
opt -load /home/anoop/LLVM/build/lib/LLVMHello.so -sgvn   < hello1.ll
```

O/P: Total number of redundant expressions detected.
Printed all redundant expressions.

Figure 7.6: Commands to run *sgvn* pass

Chapter 8

Conclusion

The LLVM framework was understood. After installation of LLVM, writing and running a sample Hello pass in LLVM was learnt. Then A simple Constant propagation Pass is written.Written Pass performs constant propagation which works fine with linear control flow. It performs propagation of constant integers only, without performing folding (if possible).

A Common Subexpression Detection pass is written(considered *ADD* instructions only and LLVM IR in SSA form in implementation of *newCSE* Pass).

Reading and understanding of *A Simple Algorithm for Global Value Numbering* is done. Finally I have implemented *A Simple Algorithm for Global Value Numbering* in LLVM Framework. I have considered small set of instructions in implementation of algorithm [6]. Implemented pass *sgvn* to detect the number of redundant expressions in benchmark SPEC CPU2006 programs.I have listed number of redundant expressions detected in SPEC CPU2006 programs.

Bibliography

[1] Getting Started: Building and Running Clang. http://clang.llvm.org/get_started.html.

[2] Getting started with the llvm system. http://llvm.org/docs/GettingStarted.html.

[3] Writing an llvm pass. http://llvm.org/docs/WritingAnLLVMPass.html.

[4] Chris Lattner. Llvm. http://www.aosabook.org/en/llvm.html/, 2002.

[5] Chris Lattner and Vikram Adve. LLVM Language Reference Manual. http://llvm.org/releases/2.7/docs/LangRef.html/, 2010.

[6] Nabizath Saleena and Vineeth K Paleri. A simple algorithm for global value numbering. arXiv:1303.1880v2, 2014.

[7] Aho A.V. Sethi R., Lam M.S. and Ullman J.D. Compilers: Principles, Techniques, and Tools. Pearson Education, 2008.