CS4090 Project
Report

Detection Of Equivalent Expressions

Submitted in partial fulfillment of
the requirements for the award of the degree of

Bachelor of Technology
in
Computer Science and Engineering

Submitted by

Roll No	Names of Students
B130165CS	Akshay Babu
B130536CS	Mekala Nimesh Reddy
B130278CS	Rahul R Menon

Under the guidance of
Dr.Saleena N

Department of Computer Science and Engineering
National Institute of Technology Calicut
Calicut, Kerala, India – 673 601

Winter Semester 2016-17

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.

Signature:
B130165CS
Akshay Babu

Signature:
B130536CS
Mekala Nimesh Reddy

Signature:
B130278CS
Rahul R Menon

Place: NIT Calicut
Date: 3^{rd} May 2017

Department of Computer Science and Engineering

National Institute of Technology Calicut

Certificate

This is to certify that this is a bonafide record of the project presented by the students whose names are given below during Winter 2016-17 in partial fulfilment of the requirements of the degree of Bachelor of Technology in Computer Science and Engineering.

Roll No	Names of Students
B130165CS	Akshay Babu
B130536CS	Mekala Nimesh Reddy
B130278CS	Rahul R Menon

Dr.Saleena N
(Project Guide)

Ms.Anu Mary Chacko
(Course Coordinator)

Date:

Acknowledgment

In keeping with the ancient tradition of respecting teachers before even God almighty, we would like to express our immense gratitude to our guide Dr Saleena N for allowing us to take up this project under her and for her constant support and motivation she gave us throughout the year.

We are grateful to Dr Saidalavi Kalady, the Head of the Department, CSED, for allowing us permission to access the facilities provided by the department that were required to successfully complete our project. We would also like to thank the members of our evaluation panel, Ms Anu Mary Chacko, Dr Subashini R and Ms Athira A B for their valuable suggestions that led to an improvement in the outcome of the project as well as our presentation skills. We also thank Ms Rekha Rama Pai for helping us out with the setting up of LLVM during the initial phase of our project.

Last but not the least we would also like to thank our family and friends, without whose unwavering support this work could not have been completed.

Akshay Babu
Mekala Nimesh Reddy
Rahul R. Menon

3^{rd} **May 2017**
National Institute of Technology Calicut

Abstract

The project deals with the detection of equivalent expressions in programs. Given an input program in three address form, data flow analysis is done on the program to detect equivalent expressions which are then provided as the output. LLVM Compiler Framework is used to first generate three-address code and then do data flow analysis on the input. The focus of the project will be on the algorithm used to detect equivalent expressions in the given input program.

Contents

List of Figures

Chapter 1

Problem Definition

Develop an algorithm that detects equivalent expressions in a program by extending the algorithm to detect equivalent variables by Alpern, Wegman and Zadeck [1].

The algorithm detects redundant expressions in the code, i.e., those expressions that evaluate to the same value. Replacing redundant expressions in a program by the value obtained after evaluating the expression once will decrease the number of computations to be performed, thereby optimizing the execution time.

The algorithm by Alpern, Wegman and Zadeck uses the Static Single Assignment form to represent code. The equivalence of variables in the code is detected by checking for congruence and dominance of variables. The algorithm used is a conservative one, i.e., all equivalent variables in the program are not detected, but the variables detected as equivalent will be truly equivalent. As part of this project, we extend the algorithm by Alpern, Wegman and Zadeck to detect general expressions, not just variables.

Chapter 2

Introduction

The project deals with the detection of equivalent expressions in a given program in SSA form. Equivalent expressions in any program may result in redundant computations which can be eliminated to improve the resulting code. The Static Single Assignment form (often abbreviated as SSA form or simply SSA) is an intermediate representation (IR) in which each variable has only one definition in the program text [3]. Subscripts distinguish each definition of variables in the SSA representation [2].

Two variables are said to be equivalent at a point p if those variables contain the same values whenever control reaches p during any possible execution of the program [1]. The algorithm given in Alpern, Wegman and Zadeck [1] detects equivalent variables and we extend it so that it works for expressions in general.

The LLVM compiler infrastructure project is a collection of modular and reusable compiler and tool chain technologies [6]. The intermediate form used by LLVM follows the Static Single Assignment form. Passes perform the transformations and optimizations that are done by the compiler [7]. We implement the algorithm as an analysis pass in LLVM.

2.1 Background

2.1.1 Static Single Assignment Form (SSA)

Static Single Assignment form is an intermediate representation in which each variable has only one definition in the program text [4]. The usage of such a variable is dominated by its definition. Subscripts distinguish each definition of variables in the SSA representation [2]. ϕ-Functions are inserted in SSA code at join points if there are assignments for same variable in dif-

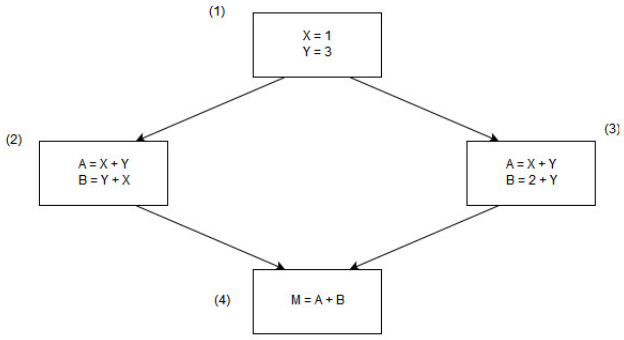

Figure 2.1: Non-SSA Code

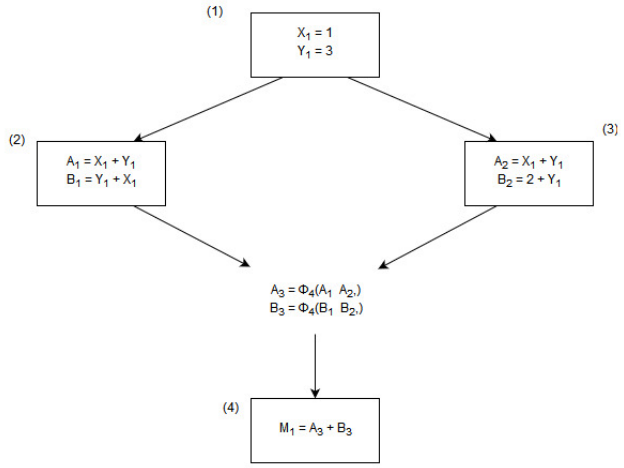

Figure 2.2: Non-SSA code in Figure 2.1 converted to SSA form

ferent branches. If a_i and a_j have defining assignments in different branches, then there would a phi-function inserted at the join point as $a_k = \phi(a_i,a_j)$.

Figure 2.1 shows the Control Flow Graph (CFG) of a sample non-SSA code. The Control Flow Graph after converting to SSA form is shown in Figure 2.2 The non-SSA code has different definitions in different branches for the same variables A and B. A has been assigned $X + Y$ in both the branches whereas B was assigned $Y * X$ in left branch while the value of $2 + Y$ was assigned to it in the right branch. In order to distinguish reaching definitions, subscripts are used in the SSA form to represent A as A_1 in the left branch and A_2 in the right branch. A similar procedure is followed for B as well. A ϕ-function is introduced at the join point in the SSA form. A_3 would be assigned either A_1 or A_2 according to the path followed while execution.

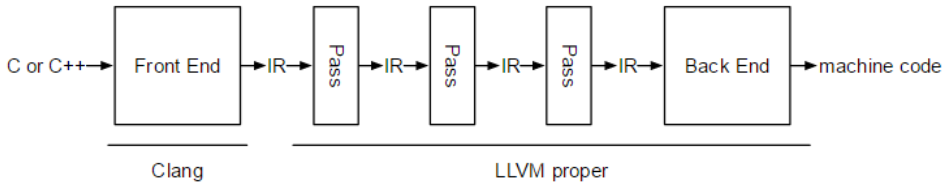

Figure 2.3: LLVM Outline [5]

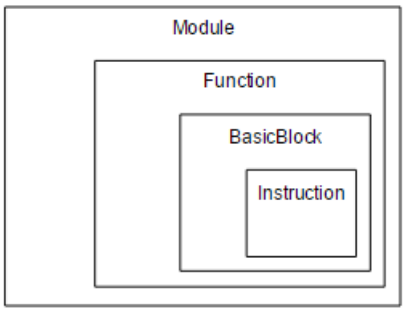

Figure 2.4: LLVM IR organization [5]

2.1.2 LLVM

The LLVM compiler infrastructure project is a collection of modular and reusable compiler and tool chain technologies [6]. The LLVM Documentation [7] describes the various libraries available in the LLVM compiler framework and the conversion of high level code into SSA form. Clang in LLVM converts the source code into Intermediate Representation (IR). LLVM IR is similar to Assembly language but is readable.

LLVM passes transform the IR to another IR in the three address form. LLVM passes may be used for analysis of the code or to transform the code input to it. LLVM has its input and output in the same IR form. All the code for analysis in this project (written in LLVM passes) is in C++.

IR organization is depicted in Figure 2.4 in brief. A module might represent source file in general. Functions are sequences of basic blocks where each basic block is a set of instructions. Further, each instruction is a single operation such as add, multiply etc.

2.1.3 Literature Survey

An introduction to compiler optimization in general is provided by Aho, Lam, Ullman and Sethi [2]. They also provide a brief description of basic blocks and control flow graphs. The book also gives examples of simple compiler optimizations like finding and replacing local common subexpressions, dead-code elimination, eliminating unreachable code, flow of control optimizations and so on. The intermediate representation used by Aho, Lam, Ullman and Sethi [2] is the three-address code form.

The Static Single Assignment intermediate representation form is described by Appel and Palsberg [3]. The criteria for equivalence of variables, dominance and equivalence is described by Muchnick [4].

An algorithm to detect equivalent variables in programs is provided by Alpern, Wegman and Zadeck [1]. It is a conservative algorithm that uses congruence and dominance of variables to detect equivalences. The algorithm works on code in the Static Single Assignment (SSA) form.

The LLVM website [6] and the LLVM documentation website [7] provide an excellent introduction to setting up LLVM and writing analysis and transformation passes in it. In our project, we have used only the LLVM analysis passes, since we are detecting the equivalent expressions and not replacing them.

Chapter 3

Design

The algorithm given by Alpern, Wegman and Zadeck detects equivalent variables in a program. It will be extended to work for expressions in general.

3.1 Algorithm to Detect equality of variables

The algorithm is described by Alpern, Wegman and Zadeck [1] and the algorithm is conservative. The algorithm consists of four stages.

Algorithm 1 Algorithm to detect equality of variables [1]

Stage 1: Building a control flow graph for the input program.
Stage 2: Converting the non-SSA code into SSA form.
Stage 3: Building a value graph for the code in SSA form.
Stage 4: Determining congruent nodes in the value graph and checking dominance.

3.1.1 Stage 1

The Input code is converted into Control Flow Graph(CFG) where each node corresponds to basic block in the program. Each edge in the CFG corresponds to a branch in the program. The multiple edges into a node are called as *Join Points*.

A CFG is built for the code in Figure 3.1 and corresponding CFG is shown in Figure 3.2 In Figure 3.2 each box represents a Basic Block and each edge connects the basic blocks and are called as *Control Flow Graph Edges*. Each Basic Block is assigned with a unique name or number.

```
if ( X < 50)
    then do
            A=2
            B=3
    end
    else do
            A=2
            B=3
    end
if ( X < Y )
    then C = 2
    else C = 3
```

Figure 3.1: Input Code

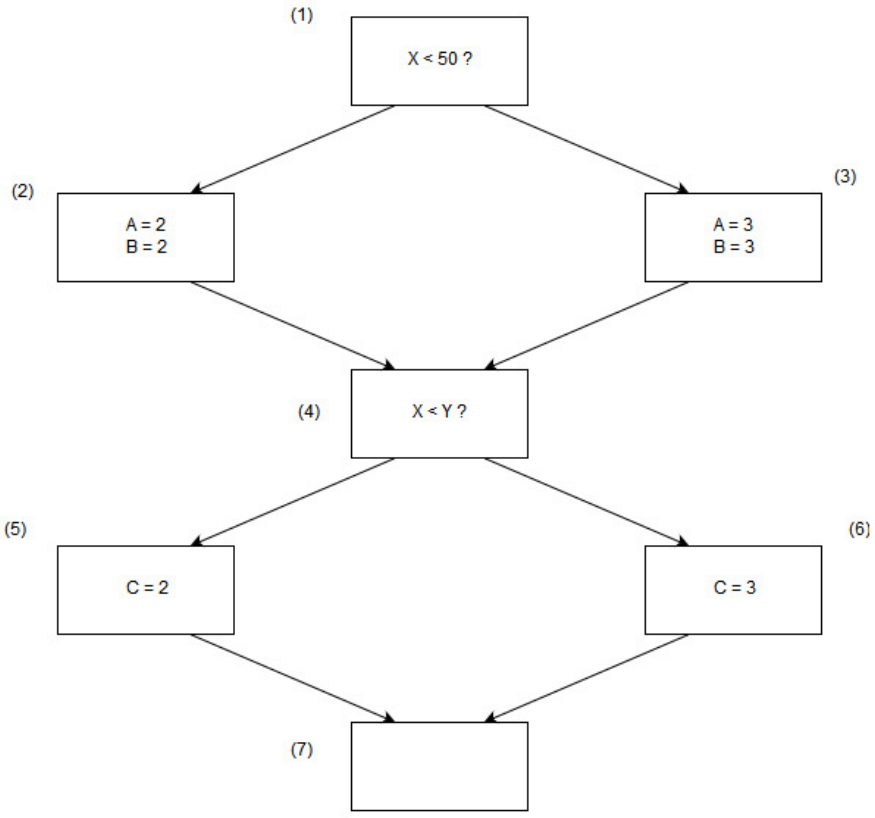

Figure 3.2: Control Flow Graph for Input code

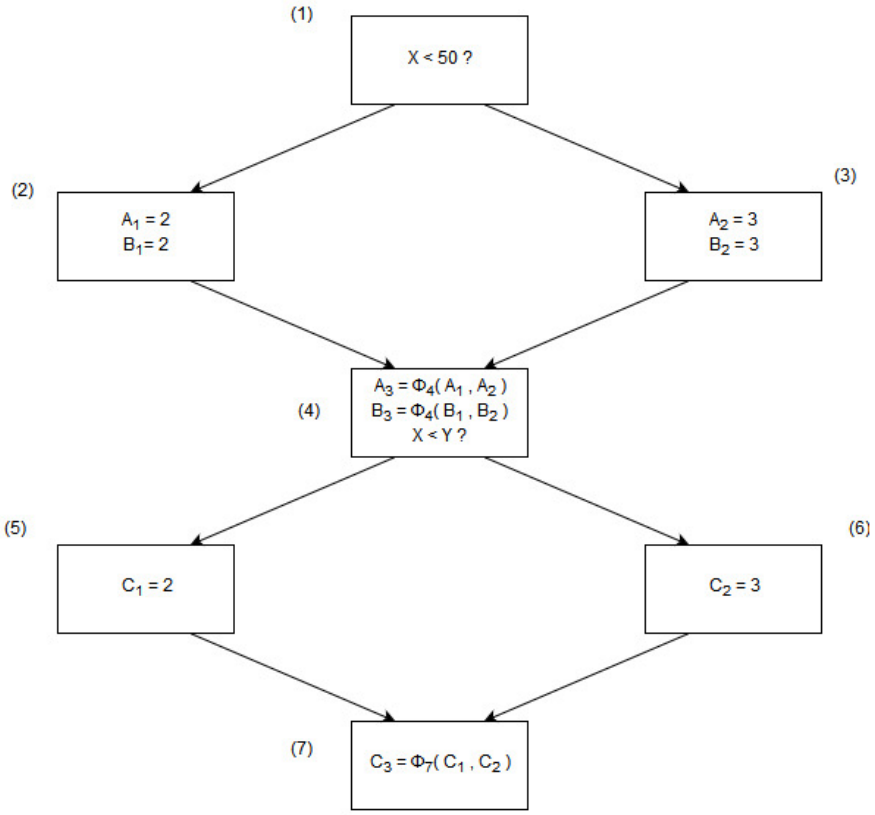

Figure 3.3: SSA for Input code

3.1.2 Stage 2

The output of previous stage would be an input for the next stage. The non-SSA code obtained in stage 1 is converted to SSA form. SSA form will have only one definition for a variable in program text. Here variables A, B in nodes 2, 3 have multiple definitions, so new variables A_1, A_2 and B_1, B_2 are introduced and ϕ-functions at their join point are introduced respectively. Similar is the case with C_1, C_2 and $C_3 = \phi_7(C_1, C_2)$.

3.1.3 Stage 3

SSA form of the input code is obtained from stage 2. It will be used as an input to build Value Graph. The value graph is a labeled, directed graph [1].The use of a variable and the assignment at which value of the graph is generated is connected by the edge of the value graph.

If the node is a function, its child nodes would be its arguments and the

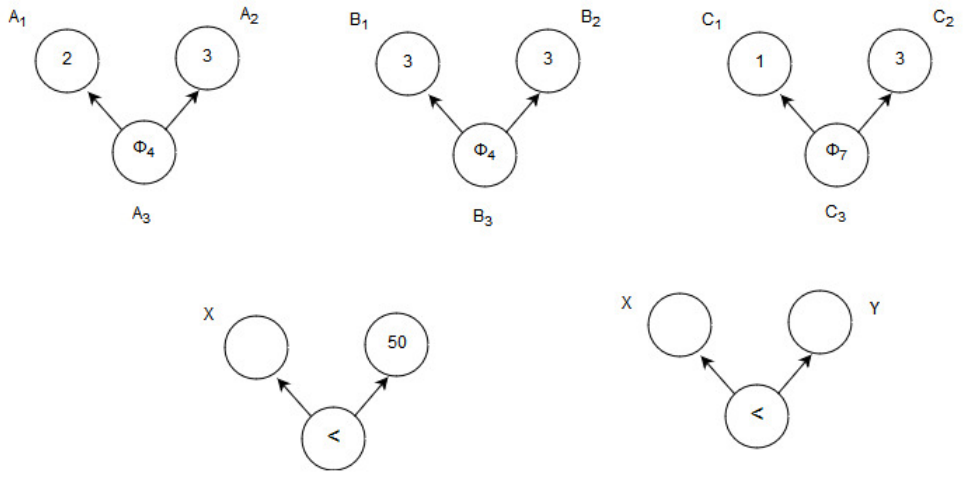

Figure 3.4: Value Graph for Input code

label would be function name.

If the node is an operator, its child nodes would be its operands and the label would be the operator.

If a node is a constant, the label would be its value and it does not have any child nodes.

The Value graph corresponding to Figure 3.2 is represented in Figure 3.4 The node with ϕ_4 as label is associated to A_3 and has two child nodes with labels 2 and 3 that are associated with A_1 and A_2 respectively. Similar is the case with B_3 and C_3.
The node with $<$ as label has operands as its children of which one is labeled as 50 (it is a constant hence not associated with any variable) and the other label is empty because of its unassigned value.

3.1.4 Stage 4

Congruence

Two nodes in the value graph are said to congruent if either:
1. They are the same node.
2. Their labels are constants and their contents are equal.
3. They have the same operators and their operands are congruent [4].

Congruence is a symmetric, reflexive and transitive relation.

Dominance

A node d dominates a node n if every path of directed edges from start to n must go through d [3].

Equivalent variables

Two variables are equivalent at a program point p, if they are congruent and if their defining assignments dominate p [1].

All the variables that are congruent will be segregated into a partition. All the variables are split into different partitions. The union of all partitions would result in entire variables in SSA code.

Partitioning

Algorithm 2 Simple Partitioning Algorithm [1]

Step 1 : *Place all nodes with the same label in the same partitions.*
Step i+1 : *Two nodes will be in the same partition in partitioning i+1 if, in partitioning i, the nodes are in the same partition and the corresponding destinations of their edges are in the same partitions.*

We begin with an initial partitioning of the nodes that puts all possibly congruent nodes in the same partition. Non congruent nodes may also start out in the same partition. We then create a new partitioning at each step of the algorithm. Each partitioning is a refinement of the previous one. In the final partitioning, two non-congruent nodes must be in different partitions.

3.2 Modified Algorithm to detect Equivalent expressions

Algorithm 3 Algorithm to detect equivalent expressions

```
for each Basic Block B_i do
    Q_i ← NULL          ▷ Initialise queue to store predecessor basic blocks

    for each predecessor p_j of B_i do
        enqueue(Q_i, p_i)

    if Q_i.length() == 0 then
        continue

    for each definition x_k = e_k in B_i do
        for each predecessor p_l of B_i in Q_i do
            for each definition y_m = e_m in p_l do
                if (x_k.getPartition() == y_m.getPartition()) then
                    enqueue(congruentQ, y_m)
                    break
        if congruentQ.length() == Q_i.length() then
            x_k = φ(congruentQ)
```

The modified algorithm detects equivalent expressions in the program. Since the SSA form of LLVM follows the three-address code format, every expression will be assigned to a temporary variable. So, in order to find out whether two expressions are equivalent, we need to check whether the corresponding temporary variables are equal. The modification proposed by us will check whether an expression exists in all the basic blocks in the same level after a branching occurs. If so, then the algorithm will check whether the same expression is used in the following basic blocks. If any such expressions are found, then they are replaced by a ϕ function that assigns it the value from the expression of the branch that is executed. Hence, equivalent expressions in the code are found out even though their corresponding variables may not be dominating a program point.

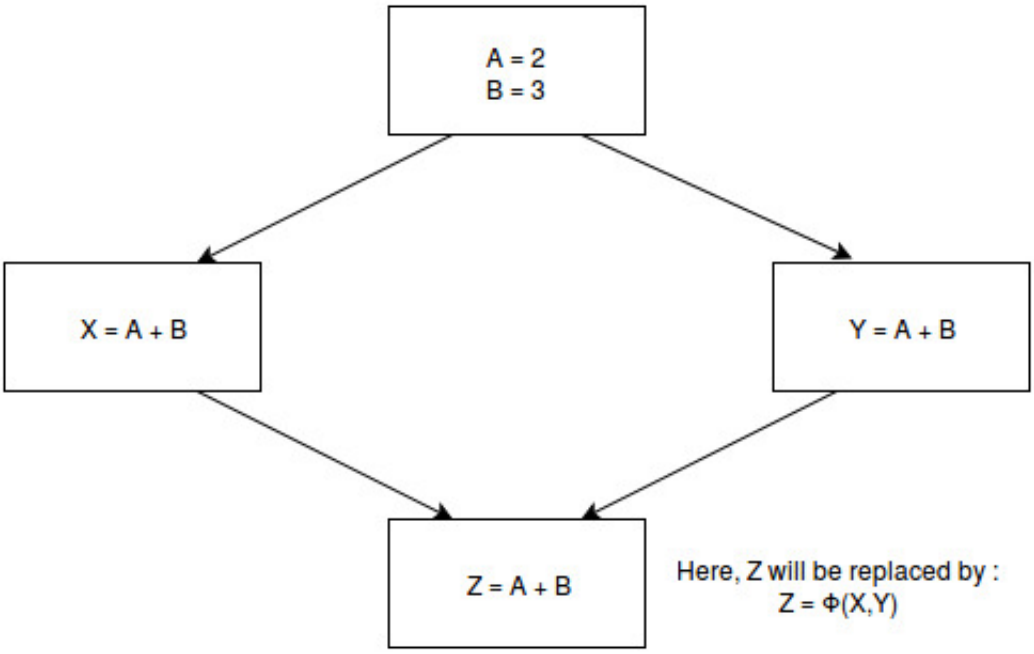

Figure 3.5: CFG using the modified algorithm

Chapter 4

Implementation

4.1 Data Structures for implementing algorithm in [1]

```
struct Node{
    string assoc;
    string label;
    list <Node*> childNodes;
    int partitionIndex;
    Instruction *inst;
};
```

A *Node* contains child nodes, its label, associated variable, *partitionIndex* as member types. If a node is a ϕ function, its child nodes will be its arguments and the label will depend on the basic block it belongs to. If the node is an operator, its child nodes will be its operands and the label will be the operator itself. If a node is a constant, the label will be its value and it does not have any child nodes. If the node is an operand, the label will be the value associated with it.

ConstNodes : list <Node>

It is a list consisting of nodes whose labels are constants and are not associated with any variable .

VGNodes : list <Node>

It is a list of all *Nodes* other than Constant nodes. They include nodes with

labels as constants, operators and ϕ functions. The union of *ConstNodes* and *VGNodes* would result in a set of all nodes.

```
struct partition{
    int index;
    list <Node*> elements;
};
```

This data structure implements a partition mentioned in Alpern [1] where each partition is distinguished by an index and contains list of nodes which are congruent.

partitionList : list <partition>

partitionList contains list of all partitions. Union of all elements of this list would result in entire nodes of the graph.

```
class ValueGraph
{
    list <Node> VGNodes;
    list <Node> ConstNodes;
    unordered_map <std::string,Node*> AssocMap;

    public:

    void insertNode(Node n);
    Node* insertConstNode(Node n);
    Node* getNode(string AssocStr);
    void displayNodes();
}
```

The above class is used to implement value graph given in [1] . The data members are *VGNodes*, *ConstNodes* and *AssocMap*. *VGNodes* is the list of all nodes in the *ValueGraph* except the constant nodes. *ConstNodes* is the list of all constant nodes in the *ValueGraph*. *AssocMap* is an unordered map mapping each associated variable to its corresponding *Node* in the *ValueGraph*. The member function *insertNode* takes a non-constant *Node* as input and inserts it into the *ValueGraph*. The member function *insertConstNode* takes a constant *Node* as input, inserts it into the *ValueGraph* and returns a pointer to it. The member function *getNode* takes an associated

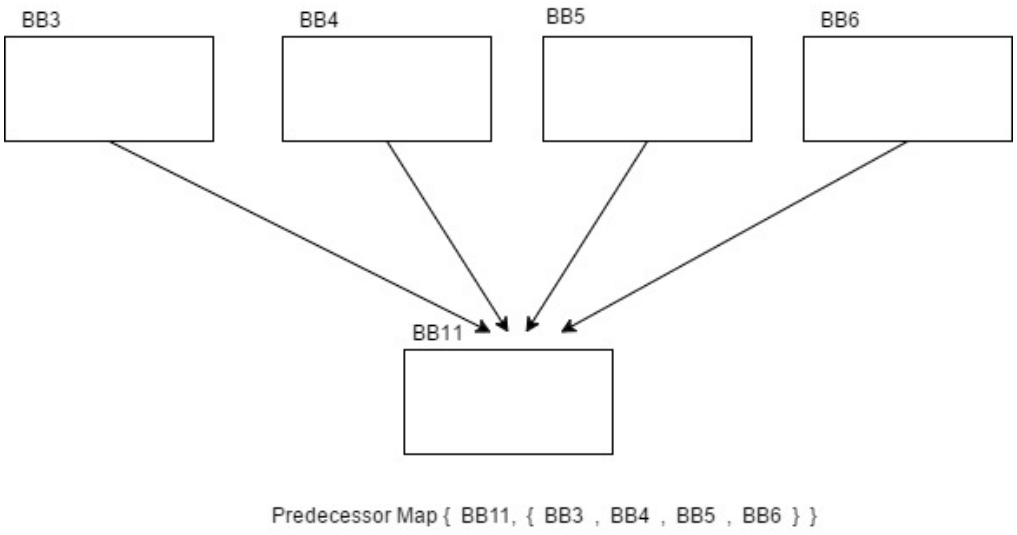

Figure 4.1: SSA for Input code

variable as input and returns the pointer to the corresponding Node. The member function *displayNodes* prints the entire *ValueGraph*.

PredecessorMap : unordered_map <std::string ,list <std::string >

PredecessorMap is a data structure which maps from string to list of strings. It maps Basic Block Name with all its predecessors. Figure 4.1 explains the implementation of *PredecessorMap* using an example.

4.2 LLVM Functions used to implement algorithm in [1]

LLVM provides lot of inbuilt functionalities to write LLVM passes. A part of those functionalities are used for implementation are as follows.

```
bool runOnFunction(Function &F)
```

runOnFunction is an inbuilt function in LLVM and can be overridden in LLVM pass. When a function in input code is encountered, this function is called and corresponding actions included in the function are applied.

```
DominatorTreeWrapperPass* DTWP = &getAnalysis<DominatorTreeWrapperPass>();
DominatorTree* DT = &DTWP->getDomTree();
```

getAnalysis is used to get *DominatorTreeWrapperPass* object. *getDomTree* member function on that object would return dominator tree. This dominator tree has a member function dominates which is used to check dominance.

Chapter 5

Results

5.1 Input

```
#include <stdio.h>
#include <stdlib.h>

int main () {
int i, j, k, l,a,b,temp;
if (i < 29) {
j = i % 77;
k = i % 7;
}
else {
k = (i % 77);
j = (i % 17) + 20;
}
a = i % 77;
b = i % 40;
if (i < 29)
l = i % 40;
else
b = i % 40;
k = i % 40;
temp = i + j + k + l;
return 0;
}
```

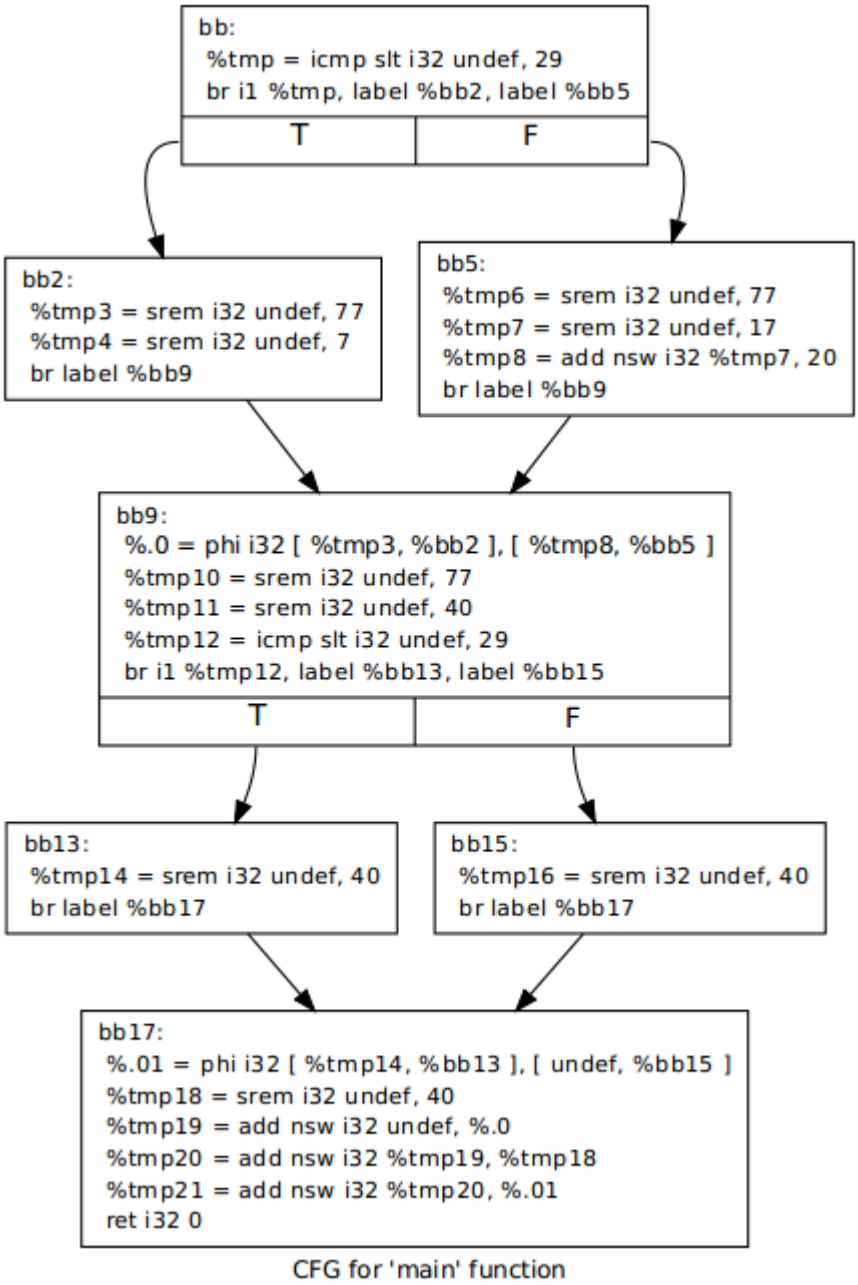

Figure 5.1: Control Flow Graph for the input

5.2 Output

5.2.1 Output on implementing the algorithm [1]

The following are equivalent:
(tmp,tmp12)
(tmp11,tmp14)
(tmp11,tmp16)
(tmp11,tmp18)

5.2.2 Output on implementing the modified algorithm

The following are equivalent:
(tmp,tmp12)
(tmp11,tmp14)
(tmp11,tmp16)
(tmp11,tmp18)

tmp18 = phi(tmp16,tmp14)
tmp10 = phi(tmp6,tmp3)

Chapter 6

Conclusion

As a part of this project, we installed and set up LLVM and ran analysis passes on it. The algorithm developed by Alpern, Wegman and Zadeck [1] was implemented using C++ as an analysis pass. It was also extended so that it works for expressions in general and not just variables.

In order to check for equivalent variables in the input code, the Value Graph data structure was implemented so that congruence of variables could be checked. Dominance of variables was checked using LLVM. The information obtained from both these steps were combined to find out equivalent variables.

An extension to the above algorithm was proposed so that equivalent expressions in the code are also detected. This was done by checking for equivalence of congruent expressions along paths in addition to checking for equivalence of congruent expressions using dominance. Currently, the algorithm works when the same version of variables is obtained along different paths. This improved algorithm was also implemented in LLVM using C++.

The future work that remains to be done is to generalize the checking for equivalence so that even if different versions of variables are obtained along different paths, equivalence of the corresponding expressions will be detected. Upon successful conclusion of this, the algorithm can be used to detect equivalent expressions and then perform an LLVM transformation pass that replaces the multiple computations by a single one to improve the efficiency of execution of the program.

References

[1] B. Alpern, M. N. Wegman, and F. K. Zadeck. 1988. Detecting equality of variables in programs. In Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '88). ACM, New York, NY, USA, 1-11. DOI = http://dx.doi.org/10.1145/73560.73561

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[3] Andrew W. Appel and Jens Palsberg. 2003. Modern Compiler Implementation in Java (2nd ed.). Cambridge University Press, New York, NY, USA

[4] Steven S. Muchnick. 1998. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[5] LLVM for Grad Students, `http://www.cs.cornell.edu/~asampson/blog/llvm.html`

[6] The LLVM Compiler Infrastructure, `http://llvm.org/`

[7] LLVM design and Overview, `http://llvm.org/docs/`