

POINTER OPTIMIZATION USING SSA BASED INTERMEDIATE REPRESENTATION FOR OPTIMIZING COMPILERS

A DISSERTATION

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

of

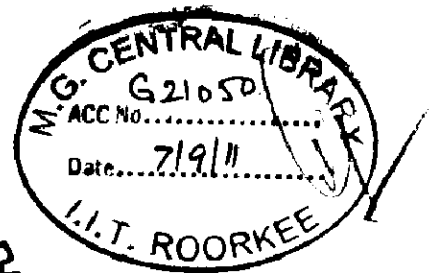
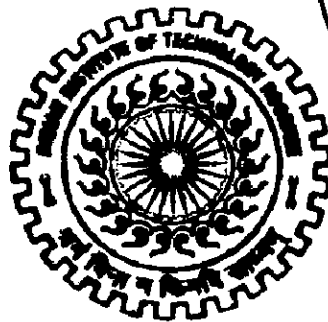
MASTER OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

By

BARHATE DEODATTA MOHAN



**DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY ROORKEE
ROORKEE -247 667 (INDIA)
JUNE, 2011**

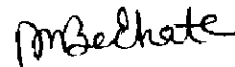
CANDIDATE'S DECLARATION

I hereby declare that the work, which is being presented in the dissertation entitled “**POINTER OPTIMIZATION USING SSA BASED INTERMEDIATE REPRESENTATION FOR OPTIMIZING COMPILERS**” towards the partial fulfillment of the requirement for the award of the degree of **Master of Technology in Computer Science and Engineering** submitted in the Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, Roorkee, Uttarakhand (India) is an authentic record of my own work carried out during the period from July 2010 to June 2011, under the guidance of **Dr. A. K. Sarje, Professor**, Department of Electronics and Computer Engineering, IIT Roorkee.

The matter presented in this dissertation has not been submitted by me for the award of any other degree of this or any other Institute.

Date: 09/06/11

Place: Roorkee



(**BARHATE DEODATTA MOHAN**)

CERTIFICATE

This is to certify that the above statement made by the candidate is correct to the best of my knowledge and belief.

Date:

Place: Roorkee



(**Dr. A. K. Sarje**)

Professor

Department of Electronics and Computer Engineering

IIT Roorkee.

ACKNOWLEDGEMENTS

First and foremost, I would like to extend my heartfelt gratitude to my guide and mentor **Dr. A. K. Sarje**, Professor, Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, for his invaluable advices, guidance, encouragement and for sharing his broad knowledge. His wisdom, knowledge and commitment to the highest standards inspired and motivated me. He has been very generous in providing the necessary resources to carry out my research. He is an inspiring teacher, a great advisor, and most importantly a nice person.

I also wish to thank Pushkar Jambhalekar for his valuable suggestions. I am greatly indebted to all my friends, who have graciously applied themselves to the task of helping me with ample moral supports and valuable suggestions.

On a personal note, I owe everything to my parents. The support which I enjoyed from my father, mother and other family members provided me the mental support I needed.

BARHATE DEODATTA MOHAN

Abstract

In optimizing compilers, most of the optimizing algorithms are applied on intermediate representation. Optimizing algorithms such as constant propagation can be applied on intermediate code using data flow analysis which is expensive regarding compilation time. Also for pointer variables, these algorithms cannot be applied directly on pointers as pointers are difficult to analyze.

In this dissertation entitled “POINTER OPTIMIZATION USING SSA BASED INTERMEDIATE REPRESENTATION FOR OPTIMIZING COMPILERS”, new approach for applying optimization algorithms such as constant propagation on pointer variables has been proposed. To avoid data flow analysis, static single assignment (SSA) form is used for intermediate representation and for pointers, alias classes are used. So using SSA and each alias class as a single variable, constant propagation algorithm can be applied on pointer variables very efficiently reducing the number of instructions in a program.

Table of Contents

Candidate's Declaration & Certificate	i
Acknowledgements	ii
Abstract	iii
Table of Contents	iv
List of Figures	vi
List of Tables	viii
1. Introduction	1
1.1 Introduction.....	1
1.2 Motivation.....	3
1.3 Statement of the Problem.....	3
1.4 Organization of the Report.....	3
2. Background and Literature Review	5
2.1 Intermediate Language.....	5
2.2 Definition of SSA.....	6
2.3 Where to place ϕ -Functions.....	7
2.4 Terminologies.....	8
2.4.1 Control Flow Graph.....	8
2.4.2 Dominance and Dominance Frontiers.....	9
2.5 Other Forms of SSA.....	9
2.5.1 Building Pruned SSA.....	9
2.5.2 Semi-pruned Form to Use Fewer ϕ -functions.....	10
2.6 Alias Analysis.....	12
2.6.1 Aliasing and Re-ordering.....	13
2.6.2 Type Based Alias Analysis.....	13
2.6.3 Flow Based Alias Analysis.....	14
2.7 Optimizations on SSA.....	14
2.7.1 Constant propagation.....	14

2.7.2	Copy Propagation.....	15
2.7.3	Global Value Numbering.....	15
2.8	Literature Review.....	16
2.9	Research Gaps.....	18
3.	Proposed Work and Implementation	19
3.1	Proposed Work.....	19
3.2	Technologies used.....	20
3.2.1	Lex and YACC.....	20
3.2.2	Grammar used to parse CFG file.....	21
3.3	Construction of SSA.....	22
3.3.1	Constructing Control flow graph from CFG specification file.....	23
3.3.2	Computing Dominance Frontier.....	23
3.3.3	Inserting ϕ -functions.....	24
3.3.4	Renaming of variables.....	24
3.3.5	CFG Specification File.....	27
3.3.6	Construction of Dominance frontier sets.....	27
3.3.7	Writing output in the file.....	28
3.4	Applying Constant Propagation Algorithm.....	30
4.	Results	34
4.1	Results after Applying Constant Propagation Algorithm.....	34
4.2	Comparison.....	35
5.	Conclusion and Future Work	37
5.1	Conclusion.....	37
5.2	Future Work.....	37
	REFERENCES.....	38
	LIST OF PUBLICATIONS.....	40

LIST OF FIGURES

Figure 1.1	Compiler Phases	1
Figure 1.2	Using SSA as IR in Compiler.....	2
Figure 2.1	Simple C Language Program.....	5
Figure 2.2	Three address Code	6
Figure 2.3	Straight Line Code and Its Static Single Assignment Version	6
Figure 2.4	If-then-else and Its Static Single Assignment Version.....	7
Figure 2.5	Illustrating where to keep Φ -function.....	7
Figure 2.6	Algorithm to compute non-local names.....	10
Figure 2.7	Simple code and it's three forms of SSA	11
Figure 2.8	Applying constant propagation on SSA	15
Figure 2.9	After applying GVN on SSA	15
Figure 2.10	If-then-else and its SSA form.....	17
Figure 2.11	Construction of alias classes for pointers in intermediate code.....	18
Figure 3.1	Applying constant propagation on SSA with Alias Classes.....	19
Figure 3.2	Optimized code.....	20
Figure 3.3	Algorithm to compute dominance frontier.....	23
Figure 3.4	Algorithm to insert ϕ -functions.....	24
Figure 3.5	Algorithm for renaming of variables.....	25
Figure 3.6	Structures computed after first module	26

Figure 3.7	Output after second module	26
Figure 3.8	C language program and it's Control Flow Graph.....	27
Figure 3.9	CFG and dominance frontier set for each node.....	28
Figure 3.10	Test1 program and its SSA form.....	31
Figure 3.11	SSA code and optimization of SSA code.....	32
Figure 4.1	Number of instructions for optimized code.....	35
Figure 4.2	% improvement for various approaches.....	36

Chapter 1

INTRODUCTION

1.1 Introduction

Compiler optimization is the process of tuning the output of a compiler to minimize or maximize some attributes of an executable computer program. The most common requirement is to minimize the time taken to execute a program; a less common one is to minimize the amount of memory occupied.

Most compilers translate the source program first to some form of intermediate representation and convert from there into machine code. The intermediate representation is a machine and language-independent version of the original source code. Although converting the code twice introduces another step, use of an intermediate representation provides advantages in increased abstraction, cleaner separation between the front and back ends, and adds possibilities for retargeting cross-compilation [4]. Intermediate representations also lend themselves to supporting advanced compiler optimizations and most optimization is done on this form of the code.

In code optimization, data structure choices directly influence the power and efficiency of program optimization. Code optimization is generally implemented using a sequence of optimizing transformations which take a program and transform it to produce an output program. While the program goes through various levels in compilation process, the optimization takes place in some levels [1].

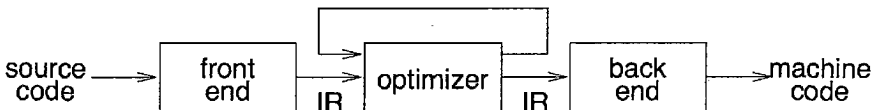


Figure 1.1: Compiler Phases

Front end produces intermediate representation (IR). The optimizer transforms IR into more efficient program and then back end transforms it to machine code. The

optimization also takes place in back end after generation of machine code but major work of optimization is done in optimizer phase. An intermediate language is used for IR.

While optimizing three address code, compiler has to perform data flow analysis which is very expensive in terms of time complexity. So static single assignment (SSA) form has been proposed to represent data flow and control flow properties of program. Static single assignment form is an intermediate representation (IR) in which every variable is assigned exactly once [5]. SSA is developed to avoid data flow analysis and to perform certain optimizations efficiently.

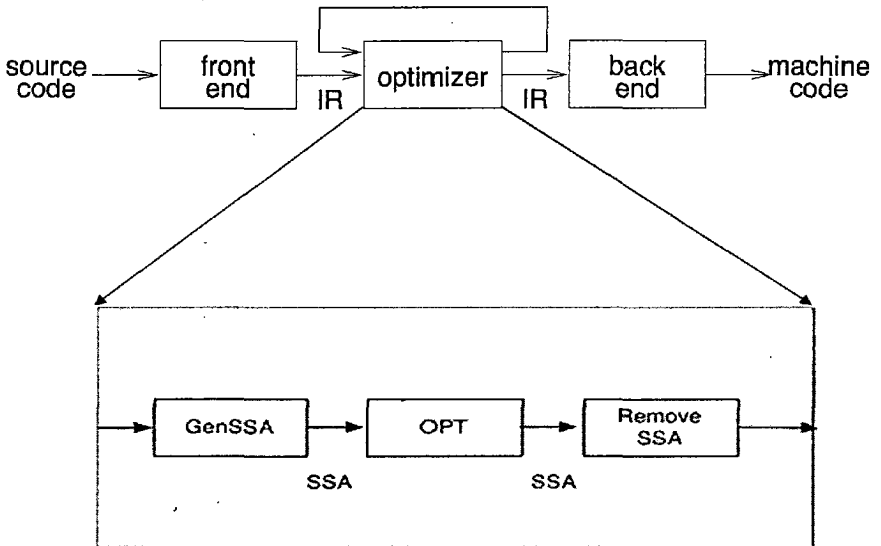


Figure 1.2: Using SSA as IR in compiler

The three address code is converted to SSA form and then after optimizing SSA form it is again translated out of SSA. SSA form has to be translated out as ϕ -functions (which are inserted in the program for control flow validation purposes) in SSA are not recognized by processor.

1.2 Motivation

Static Single-Assignment form is an intermediate format that allows optimizations to be done efficiently and easily. Every variable receives exactly one assignment during its lifetime, and ϕ -functions are added at places where program flow joins. Static single assignment form has been proposed to represent data flow and control flow properties of program [17].

SSA simplifies some optimization algorithms like constant propagation. Whenever we apply constant propagation algorithm on SSA code, the occurrence of a variable is replaced by its value directly since there is only one definition of that variable [6]. But we cannot apply constant propagation algorithm directly on pointer variables since there may be two or more variables pointing to same memory location. Any variable changing the value at that location changes the value of other variables. For pointers, we have to use alias classes. Each alias class represents the set of variables pointing to same memory location.

1.3 Problem Statement

In this dissertation work, we propose an approach for pointer optimization in C program using SSA based intermediate representation. We use SSA as intermediate representation to avoid data-flow analysis. As it is not possible to apply constant propagation algorithm on pointer variables directly, we use alias classes for pointers. Whenever any variable in alias class is changed, the alias class is also renamed.

1.4 Organization of the report

Chapter 2 gives the basics of SSA and how it can be used as intermediate representation. This chapter also explains how SSA simplifies some optimizing algorithms. Various types of SSA are also explained. Literature review and research gaps are discussed.

Chapter 3 explains proposed work for pointer optimization. It further explains construction of SSA and alias classes and how to apply constant propagation algorithm on SSA.

Chapter 4 presents results gathered after applying optimization algorithm on some test programs. It further explains comparison between various approaches of applying constant propagation algorithm on SSA.

Chapter 5 presents the conclusions of the work and suggests future work that can be done to extend the work.

Chapter 2

BACKGROUND AND LITERATURE REVIEW

2.1 Intermediate Language

An intermediate language is the language of an abstract machine designed to aid in the analysis of computer programs. Compiler first translates the source code of a program into a form more suitable for code-improving transformations, as an intermediate step before generating object or machine code for a target machine.

A popular format for intermediate language is three address code. Each instruction in three address code can be described as:

$$result := operand1 operator operand2$$

such as:

$$x := y op z$$

where x , y and z are variables, constants or temporary variables generated by the compiler. op represents any operator, e.g. an arithmetic operator.

Expressions containing more than one fundamental operation, such as:

$$p := x + y \times z$$

are not representable in three-address code as a single instruction. Instead, they are decomposed into an equivalent series of instructions, such as

$$t_1 := y \times z$$

$$p := x + t_1$$

The term three-address code is still used even if some instructions use more or fewer than two operands. The key features of three-address code are that every instruction implements exactly one fundamental operation, and that the source and destination may refer to any available register. Consider following C language program.

```
int main(void)
{
    int i;
    int b[10];
    for (i = 0; i < 10; ++i)
    {
        b[i] = i*i;
    }
}
```

Figure 2.1 : Simple C language program

The preceding C program, translated into three-address code, might look something like the following:

```

        i := 0           ; assignment
L1:     if i >= 10 goto L2 ; conditional jump
        t0 := i*i
        t1 := &b        ; address-of operation
        t2 := t1 + i    ; t2 holds the address of b[i]
        *t2 := t0       ; store through pointer
        i := i + 1
        goto L1
L2:

```

Figure 2.2 : Three address code

2.2 Definition of SSA

A procedure is in static single-assignment form if every variable assigned a value in it occurs as the target of only one assignment [5]. The static single assignment form of a program provides data flow information in a form which makes some compiler optimizations easy to perform. A use of a variable may use the value produced by a particular definition if and only if the definition and the use have exactly the same name for the variable in the SSA form of the procedure. This simplifies and makes more effective several kinds of optimizing transformations. Thus, it is valuable to be able to translate a given representation of a procedure into SSA form.

As shown in figure 2.3, each assignment to a variable is given a unique name and all of the uses reached by that assignment are renamed to match the assignment's new name.

$V \leftarrow 4$	$V_1 \leftarrow 4$
$X \leftarrow V + 5$	$X_1 \leftarrow V_1 + 5$
$V \leftarrow 8$	$V_2 \leftarrow 8$
$Y \leftarrow V + 3$	$Y_1 \leftarrow V_2 + 3$
(a).Straight line code	(b).SSA form

Figure 2.3: Straight line code and its static single assignment version

Most programs, however, have branch and join nodes. At the join nodes, a special function called ϕ -function is inserted. ϕ -function selects any one of its arguments as per the control transfers from one node to another node. ϕ -functions are there for validation purpose. They are not recognized by processor. So after performing optimization on SSA code, it has to be transformed out from SSA [3]. ϕ -functions must be removed.

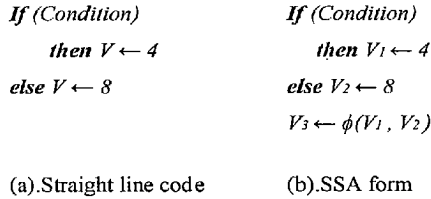


Figure 2.4: if-then-else and its static single assignment version

2.3 Where to place ϕ -Functions

At first glance, careful placement might seem to require the enumeration of pairs of assignment statements for each variable. Checking whether there are two assignments to V that reach a common point might seem to be intrinsically nonlinear [11].

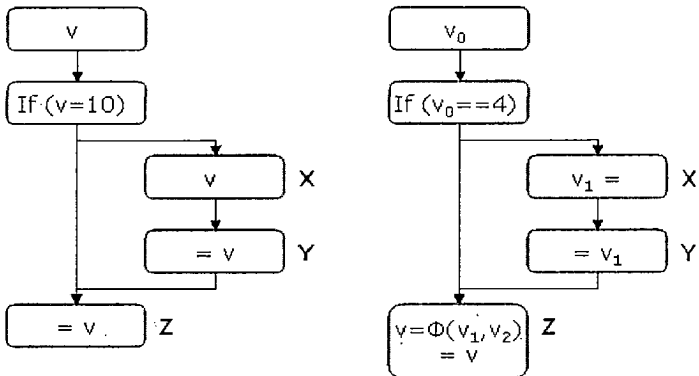


Figure 2.5: Illustrating where to keep Φ -function

As shown in figure 2.5, suppose that a variable V has just one assignment in the original program, so that any use of V will be either a use of the value V_0 at entry to the program or a use of the value V_1 from the most recent execution of the assignment to V . Let X be the basic block of code that assigns to V , so X will determine the value of V when control flows along any edge $X \rightarrow Y$ to a basic block Y . When entered along $X \rightarrow Y$, the code in Y will see V_1 , but all paths to Y must still go through X (in which case X is said to strictly dominate Y), then the code in Y will always see V_1 . Indeed, any node strictly dominated by X will always see V_1 , no matter how far from X it may be.

Eventually, however, control may be able to reach a node Z not strictly dominated by X . Z sees V_1 along one in-edge but may see V_0 along another in-edge. Then Z is said to be in the dominance frontier of X and is clearly in need of a ϕ -function for V . In general, no matter how many assignments to V may appear in the original program and no matter how complex the control flow may be, ϕ -functions for V can be placed by finding the dominance frontier of every node that assigns to V .

2.4 Terminologies

Construction of SSA involves recognizing where to place ϕ -functions and renaming of variables. To recognize location of ϕ -functions the knowledge of dominance frontiers of nodes in control flow graph is needed. Some of the terminologies used in the construction of SSA are given below

2.4.1 Control Flow Graph

A control flow graph (CFG) in computer science is a representation, using graph notation, of all paths that might be traversed through a program during its execution. The statements of a program are organized into basic blocks, where program flow enters a basic block at its first statement and leaves the basic block at its last statement [15]. A control flow graph is a directed graph whose nodes are the basic blocks of a program and two additional nodes, Entry and Exit.

2.4.2 Dominance and Dominance Frontiers

Let X and Y be nodes in the control flow graph(CFG) of a program. If X appears on every path from Entry to Y , then X *dominates* Y [8]. If X dominates Y and $X \neq Y$, then X *strictly dominates* Y . Here, $X \gg Y$ indicates strict domination and $X \geq Y$ indicates domination. If X does not strictly dominate Y , then $X \not\gg Y$ is written. The dominance frontier $DF(X)$ of a CFG node X is the set of all CFG nodes Y such that X dominates a predecessor of Y but does not strictly dominate Y .

$$DF(X) = \{ Y \mid (\exists P \in Pred(Y)) (X \geq P \text{ and } X \not\gg Y) \}$$

2.5. Other forms of SSA

2.5.1 Building Pruned SSA

Minimal SSA form relies entirely on dominator information to determine where to insert ϕ -functions [11]. The dominance frontier correctly captures the potential flow of values, but ignores the data-flow facts like knowledge about the lifetimes of values gleaned from analyzing their definitions and uses. Because of this, the minimal SSA construction will insert a ϕ -node for V at join point where V is not live.

To improve on minimal SSA, the compiler first performs “liveness analysis” on the routine [7]. Liveness analysis produces, for each block, a set of values that are live on entry to the block [18]. The actual construction of pruned SSA is quite similar to the construction of minimal SSA. The minimal SSA construction inserts a ϕ -node for V in every node $n \in DF^+(A(V))$. The pruned SSA construction changes this to insert a ϕ -node for V in every node $n \in DF^+(A(V))$, where $V \in LIVE_IN(n)$. These changes can drastically reduce the number of ϕ -nodes.

The pruned SSA construction algorithm costs more than the minimal SSA construction. Building liveness analysis increases compilation time and space requirements for building SSA [7]. The space requirement increases, since each block has a number of large sets associated with it. These larger memory requirements can directly degrade performance. So pruned SSA is not used practically to construct SSA for intermediate representation.

2.5.2 Semi-pruned Form to Use Fewer ϕ -functions

Minimal SSA form places ϕ -nodes by looking only at the dominance frontier without regard to liveness. It is possible that a ϕ -node will be inserted for a name which is not subsequently used. These extra ϕ -nodes waste space and time. Pruned SSA form relies on liveness analysis to ensure that no such dead ϕ -nodes are inserted. Since the pruned form relies on additional analysis, it may be slower to build.

The third form of SSA called semi-pruned SSA is developed. The speed and space advantage of this form over the other two relies on the observation that many names in a routine are defined and used wholly within a single basic block [14]. For example, the compiler typically generates temporary names to hold intermediate steps; these compiler-generated names often have short lifetimes. Semi-pruned SSA capitalizes on this observation by computing the set of names that are live on entry to some basic block in the program. These are called “non-local” names. The construction algorithm computes dominance frontiers only for non-local names. Therefore, the number of ϕ -nodes will lie between that of the minimal and pruned forms, but the non-local names are much cheaper to compute than full-blown liveness analysis. Therefore, semi-pruned form represents a compromise between the time required to perform liveness analysis and the reduction in the number of ϕ -nodes that it allows.

The algorithm to discover the non-local names is as follows:

```
non-locals  $\leftarrow \emptyset$ 
for each block B
    killed  $\leftarrow \emptyset$ 
    for each instruction  $v \leftarrow x \text{ op } y$  in B
        if x  $\notin$  killed then
            non-locals  $\leftarrow$  non-locals  $\cup \{x\}$ 
        if y  $\notin$  killed then
            non-locals  $\leftarrow$  non-locals  $\cup \{x\}$ 
        killed  $\leftarrow$  killed  $\cup \{v\}$ 
```

Figure 2.6: Algorithm to compute non-local names

As per the algorithm, the compiler makes a simple forward pass over each basic block. When it discovers an operand that has not already been defined within the block (the killed set), it must be a non-local name. This is simpler than performing the complete live analysis required for pruned SSA construction. Computing non-local names requires just two sets, non-local and killed. The time and space requirements for building non-local are, therefore minimal.

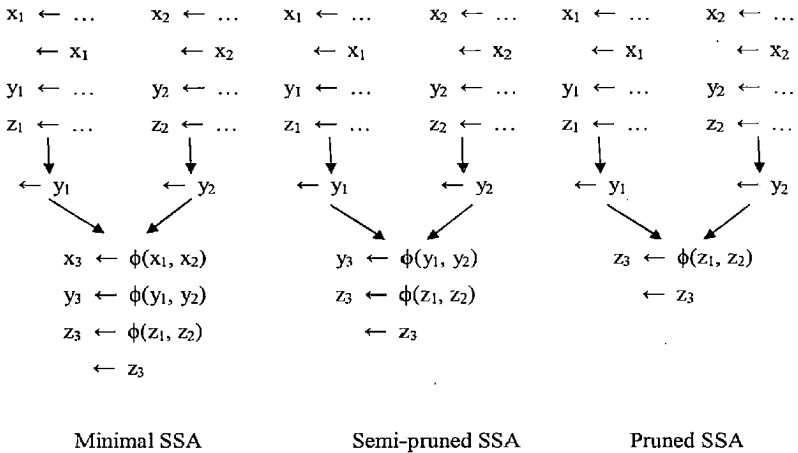
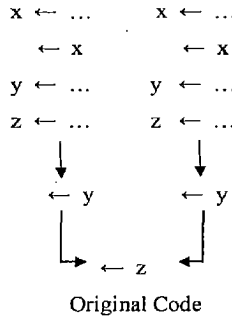


Figure 2.7 : Simple code and it's three forms of SSA

Figure 2.7 illustrates the differences between the three forms of SSA. In the original code, the three variables, x , y , and z are defined. The three graphs at the bottom of the figure compare the ϕ -nodes which the three forms of SSA insert. The minimal SSA form contains ϕ -nodes for all three variables. Clearly, the ϕ -nodes for x and y are unnecessary; these variables are never used again. The semi-pruned SSA form does not contain a ϕ -node for x because it is not live across any basic block boundary. However, for y , a ϕ -node is inserted, because it is live across some block boundary, and that is the limit of analysis used. The pruned SSA form contains a ϕ -node for z only.

2.6 Alias Analysis

Alias analysis is a technique in compiler theory, used to determine if a storage location may be accessed in more than one way [9]. Two pointers are said to be aliased if they point to the same location. Alias analysis techniques are usually classified by flow-sensitivity and context-sensitivity. They may determine may-alias or must-alias information. The term alias analysis is often used interchangeably with term points-to analysis, a specific case.

In general, alias analysis determines whether or not separate memory references point to the same area of memory [16]. This allows the compiler to determine what variables in the program will be affected by a statement. For example, consider the following section of code that accesses members of structures:

```
*p = 1;
*q = 2;
i = *p + 3;
```

There are three possible alias cases here:

1. The variables p and q cannot alias.
2. The variables p and q must alias.
3. It cannot be conclusively determined at compile time if p and q alias or not.

If p and q cannot alias, then $i = *p + 3$; can be changed to $i = 4$. If p and q must alias, then $i = *p + 3$; can be changed to $i = 5$. In both cases, we are able to perform optimizations from the alias knowledge. On the other hand, if it is not known if p and q alias or not, then no optimizations can be performed and the whole of the code must

be executed to get the result. Two memory references are said to have a *may-alias* relation if their aliasing is unknown.

2.6.1 Aliasing and Re-ordering

Aliasing introduces strong constraints on program execution order. If two write accesses which alias occur in sequence in a program text, they must occur in sequence in the final machine code. Re-ordering the accesses will produce an incorrect program result (in the general case) [10]. The same is true for a write access and a read access.

However, if two read accesses which alias occur in sequence in a program text, they need not occur in the same sequence in the machine code - aliasing read accesses are safe to re-order. This is because the underlying data is not changed by the read operation, so the same value is always read. It is vitally important that a compiler can detect which accesses may alias each other, so that re-ordering optimizations can be performed correctly.

In alias analysis, we divide the program's memory into *alias classes*. Alias classes are disjoint sets of locations that cannot alias to one another. For the discussion here, it is assumed that the optimizations done here occur on a low-level intermediate representation of the program. This is to say that the program has been compiled into binary operations, jumps, moves between registers, moves from registers to memory, moves from memory to registers, branches, and function calls/returns.

2.6.2 Type Based Alias Analysis

If the language being compiled is type safe, the compiler's type checker is correct, and the language lacks the ability to create pointers referencing local variables, (such as ML, Haskell, or Java) then some useful optimizations can be made. There are many cases where we know that two memory locations must be in different alias classes:

1. Two variables of different types cannot be in the same alias class since it is a property of strongly typed, memory reference-free (i.e. references to memory locations cannot be changed directly) languages that two variables of different types cannot share the same memory location simultaneously.

2. Allocations local to the current stack frame cannot be in the same alias class as any previous allocation from another stack frame. This is the case because new memory allocations must be disjoint from all other memory allocations.
3. Each record field of each record type has its own alias class, in general, because the typing discipline usually only allows for records of the same type to alias. Since all records of a type will be stored in an identical format in memory, a field can only alias to itself.
4. Similarly, each array of a given type has its own alias class.

When performing alias analysis for code, every load and store to memory needs to be labeled with its class. We then have the useful property, given memory locations A_i and B_j with i, j alias classes, that if $i = j$ then A_i may-alias B_j , and if $i \neq j$ then the memory locations will not alias.

2.6.3 Flow Based Alias Analysis

Analysis based on flow, unlike type based analysis, can be applied to programs in a language with references or type-casting [10]. Flow based analysis can be used in lieu of or to supplement type based analysis. In flow based analysis, new alias classes are created for each memory allocation, and for every global and local variable whose address has been used. References may point to more than one value over time and thus may be in more than one alias class. This means that each memory location has a set of alias classes instead of a single alias class.

2.7 Optimizations on SSA

SSA simplifies many compiler optimization algorithms as every variable in SSA has only one definition simplifying data flow analysis [13]. Some of the optimizations enhanced by SSA are as:

2.7.1 Constant propagation

SSA form simplifies constant propagation: whenever a definition of the form $x \leftarrow c$, where c is a constant, is encountered, then all uses of x can be replaced by c . Moreover, the definition itself can be deleted from the program, as it is now dead.

$b := 3$	$b_1 := 3$
$c := 1 + b$	$c_1 := 1 + 3$
$b := 4$	$b_2 := 4$
$d := b + c$	$d_1 := 4 + c_1$
Original code	SSA

Figure 2.8 : Applying constant propagation on SSA

So in the original code analysis for variable b is needed for constant propagation as its value is changed in code. But as in SSA, after renaming every variable is assigned only once, analysis not needed. Thus, simplifying the optimization.

2.7.2 Copy Propagation

Copy propagation can be handled in a similar fashion as constant propagation: definition of the form $x \leftarrow y$ can be deleted, and all uses of x replaced by uses of y without analysis.

$x_1 := y_1$	$x_1 := y_1$
$c_1 := 1 + x_1$	$c_1 := 1 + y_1$
$d_1 := x_1 + c_1$	$d_1 := y_1 + c_1$
Before replace	After replace

Here, after replacing x_1 by y_1 , x_1 can be deleted.

2.7.3 Global Value Numbering

Global Value Numbering (GVN) is a compiler optimization based on the SSA intermediate representation. It sometimes helps eliminate redundant code that common subexpression elimination (CSE) does not.

$w_1 := 3$	$w_1 := 3$
$x_1 := 3$	$x_1 := w_1$
$y_1 := x_1 + 4$	$y_1 := w_1 + 4$
$z_1 := w_1 + 4$	$z_1 := y_1$
SSA	After GVN

Figure 2.9 : After applying GVN on SSA

A GVN routine would assign the same value number to w and x , and the same value number to y and z . For instance, the map $w_1 \leftarrow 1, x_1 \leftarrow 1, y_1 \leftarrow 2, z_1 \leftarrow 2$ would constitute an optimal value number mapping for this block. The reason that GVN is more powerful than CSE comes from the fact that CSE matches lexically identical expressions whereas the GVN tries to determine an underlying equivalence. For instance, in the code:

```
a := c x d
e := c
f := e x d
```

CSE would not eliminate the recomputation assigned to f , but a GVN algorithm should discover and eliminate this redundancy. SSA form is required to perform GVN.

After performing optimizations, the code in SSA form must be transformed out of SSA form as processor cannot recognise ϕ -functions in the SSA form. So ϕ -functions must be removed from code without violating the correctness of the program. The algorithms are developed for destruction of ϕ -functions.

2.8 Literature Review

The first approach to apply SSA on intermediate representation and avoid data flow analysis is due to Cytron et al[2]. They constructed an algorithm to compute SSA using the concept of dominance frontiers explained in section 3.3.2. Much of the subsequent work was based on this technique.

Bilardy and Pingali have constructed algorithms to compute SSA efficiently[11]. Using these algorithms, the number of Φ -functions can be reduced and then the processing time of applying optimization algorithms reduces. Φ -functions are special functions in SSA inserted at branches and join nodes of the program. Φ -function selects any one of its arguments as the control transfers from one node to another in control flow graph[12]. As shown in figure 2.10, after if-else, variable V can have any value depending on the *Condition* in if loop. So Φ -function is necessary for control validation.

<i>if</i> (Condition) then $V \leftarrow 4$ <i>else</i> $V \leftarrow 8$	<i>if</i> (Condition) then $V_1 \leftarrow 4$ <i>else</i> $V_2 \leftarrow 8$ $V_3 \leftarrow \Phi(V_1, V_2)$
--	---

Fig. 2.10: If-then-else and its SSA form

Lenart et al show how to apply constant propagation algorithm on SSA code[13]. While applying constant propagation algorithm on SSA, the occurrence of a variable is replaced by its value and after that its definition is deleted from the program. SSA form simplifies constant propagation since whenever a definition of the form $x \leftarrow c$, where 'c' is a constant, is encountered, and then all uses of x can be replaced by 'c'. Moreover, the definition itself can be deleted from the program, as it is now dead[7]. As shown below, b has changed its value twice, so in normal code, analysis is needed but in SSA, as it is renamed twice, there are two separate variables for b. So any use of b_1 or b_2 is replaced by its definition.

$b_1 = 3$	$b_1 = 3$
$c_1 = 1 + b_1$	$c_1 = 1 + 3$
$b_2 = 4$	$b_2 = 4$
$d_1 = b_2 + c_1$	$d_1 = 4 + c_1$
SSA code	After constant propagation

In original code without SSA, analysis for variable 'b' is needed for constant propagation as its value is changed in code. But as in SSA, after renaming every variable is assigned only once, analysis not needed. Also, variables b_1 and b_2 will be deleted after they are replaced.

Chase and Wegman propose an approach for pointer analysis using alias classes[10]. Sassa et al. apply constant propagation algorithm on pointers using alias classes[14]. This work does not involve SSA code.

Fig. 2.12 shows alias classes for pointer variables in bracket. The right hand side value is the value stored at the memory location pointed by pointer. Here, in original code, 'a' cannot be replaced by its value in the assignment to 'b' since pointer 'p' points to 'a'. So alias class for 'p' and 'a' is constructed and the value corresponding to that memory location is assigned to that alias class. Whenever the occurrence of any variable in alias class takes place, it is replaced by value assigned to

that alias class[9]. But in this approach, data flow analysis is needed as SSA is not used.

<code>int a = 3</code>	<code>int a = 3</code>
<code>int *p = &a</code>	<code>int *p = &a</code>
<code>*p = 4</code>	<code>(*p, a) = 3</code>
<code>b = a + 5</code>	<code>(*p, a) = 4</code>
	<code>b = 4 + 5</code>

Fig. 2.11: Construction of alias classes for pointers in intermediate code

2.9 Research Gaps

While applying optimizing algorithm like constant propagation or copy propagation on SSA code, the algorithm cannot be applied directly on pointer variables. If it is applied as per normal variables, the output will be incorrect. Also if we use alias classes for pointer variables, it involves data-flow analysis and applying data-flow analysis involves more time and space requirements compare to convert program into SSA code.

Chapter 3

PROPOSED WORK AND IMPLEMENTATION

3.1 Proposed Work

So far in applying constant propagation algorithm on pointers, either SSA is used leaving pointers out of it or using data flow analysis with alias classes for pointers. But alias classes and SSA are never used combined.

In our approach, we first design alias classes for pointer variables and then apply SSA on alias classes considering each alias class as a single variable. Whenever any variable in alias class is changed, the alias class is renamed and the occurrences of variables are also renamed as per the alias class renaming. Then while applying constant propagation algorithm on this code, occurrences of variables are replaced by the value of alias class if the variable is a pointer variable or pointed by a pointer variable. Otherwise normal SSA and constant folding algorithm are applied on the variable.

int a = 3	int a ₁ = 3	int a ₁ = 3
int *p = &a	int *p ₁ = &a ₁	int *p ₁ = &a ₁
b = a + 4	(*p ₁ , a ₁) = 3	(*p ₁ , a ₁) = 3
*p = 5	b ₁ = a ₁ + 4	b ₁ = 3 + 4
c = a + 6	*p ₂ = 5	*p ₂ = 5
d = *p + 2	(*p ₂ , a ₂) = 5	(*p ₂ , a ₂) = 5
c ₁ = a ₂ + 6	c ₁ = 5 + 6	
d ₁ = *p ₂ + 5	d ₁ = 5 + 5	
IR	SSA with Alias Classes	After constant propagation

Fig. 3.1: Applying constant propagation on SSA with Alias Classes

As shown in figure 3.1, pointer 'p' points to variable 'a'. So whenever any occurrence of 'a' occurs, it cannot be directly replaced by its value since pointer 'p' may also change the value. So constant propagation algorithm cannot be applied directly on 'a'. So alias class for 'p' and 'a' is designed and unique value is assigned

to that alias class. Whenever any variable in alias class is occurred at right hand side of assignment, it is replaced by the value assigned to alias class.

Here, after applying constant propagation algorithm on SSA code and replacing the variables by their value, the variables which are dead (here 'a' and 'p'), are deleted. After deleting the dead variables the code becomes,

$$b_1 = 3 + 4$$

$$c_1 = 5 + 6$$

$$d_1 = 5 + 5$$

Fig. 3.2: Optimized code

If we compare the code in figure 3.2 with original code in figure 3.1, there are less instructions compared to original code and also while compiling the program, compiler does not have to access memory for one pointer variable. This reduces execution time to large extent. Even some time is required to convert intermediate code to SSA form with alias classes and then to apply constant propagation algorithm, this approach is still very useful since these transformations take place at compile time. Only once, while compiling the program, this time is required. Afterward, while executing the program, it will take less time compare to the execution of original program.

In this approach, the interprocedural analysis is not considered as it is very complex regarding the pointer variables. Interprocedural analysis involves function calls, passing pointer variable from one function to another[15].

3.2 Technologies used

3.2.1 Lex and YACC

Lex and YACC are tools used in compiler generation. Lex generates lexical analyzer for the compiler. It creates tokens for syntax analyzer. YACC generates syntax analyzer to parse the given input file. The new grammar (other than C language grammar) is required for YACC because the CFG file generated by gcc has separate syntax compared to original C language program.

3.2.2 Grammar used to parse CFG file

In implementation of conversion of C program into SSA, CFG specification file is used (discussed in later sections). Normal C language grammar cannot be applied to this code since it contains information of nodes corresponding to basic blocks in the program. So separate grammar is developed to parse the CFG file. The grammar is as shown:

```
program : functions
;
functions : function
| function functions
;
function : func_head '{' body '}'
;
func_head : identifier '(' func_head_param ')'
| identifier '(' ' ' ')'
;
body : declaration blocks
;
declaration : decl
| decl declaration
;
decl : TYPE identifier
;
blocks : block
| block blocks
;
block : BLOCK PRED statements SUCC
;
func_head_param : identifier ',' func_head_param
| identifier
;
param : identifier ',' param
| identifier
;
statements : statement
| statement statements
;
statement : assignment
| goto_stmt
| if_stmt
| func_call
```

```

| return_stmt
| comp_stmt
;
assignment : identifier '=' identifier OP identifier
| identifier '=' identifier
;
goto_stmt : GOTO
;
if_stmt : IF '(' statements ')' goto_stmt
| if_stmt ELSE goto_stmt
;
return_stmt : RETURN
| RETURN identifier
;
comp_stmt : identifier REL_OP identifier
;
func_call : identifier '(' param ')'
| identifier '(' ')'
;
identifier : ID
| CONST
| STRING_LITERAL
| pointer identifier
| '&' identifier
| identifier '[' identifier ']'
| identifier '(' param ')'
| identifier '(' ')'
| identifier '{' '}'
| identifier '{' identifier '}'
| '{' '}'
| '{' identifier '}'
;
pointer : '*'
| '*' pointer
;

```

Above grammar is different compare to normal C language grammar in a sense that it contains only if-else loop and also all the statements are in a basic block.

3.3 Construction of SSA

Overview of the SSA Algorithm - Translation to minimal SSA form is done in four steps :

- First the control flow graph is constructed from cfg specification file generated by gcc.
- The dominance frontier mapping is constructed from the control flow graph.
- Using the dominance frontiers, the locations of the ϕ -functions for each variable in the original program are determined.
- The variables are renamed by replacing each mention of an original variable V by an appropriate mention of a new variable V_i .

3.3.1 Constructing Control flow graph from CFG specification file

Instead of constructing cfg directly from input program, the cfg is constructed using cfg specification file generated by gcc(gnu compiler collection) [5]. GCC generates cfg file for a given input file by using command 'gcc -fdump-tree-cfg file.c'.

3.3.2 Computing Dominance Frontier

Dominance frontiers capture the precise places at which Φ -functions are needed [2]. If the node A defines a certain variable, then that definition and that definition alone (or redefinitions) will reach every node A dominates.

```

for each node b
    if the number of immediate predecessors of b  $\geq 2$ 
        for each p in immediate predecessors of b
            runner := p
            while runner  $\neq$  doms(b)
                add b to runner's dominance frontier set
                runner := doms(runner)

```

Figure 3.3 : Algorithm to compute dominance frontier

Here, in figure 3.3, an immediate predecessor of node n is any node from which control is transferred to node n , and $\text{doms}(b)$ is the node that immediately dominates node b (a singleton set).

3.3.3 Inserting ϕ -functions

The algorithm in figure 3.4 inserts ϕ -functions. The outer loop of this algorithm is performed once for each variable V in the program. Several data structures are used:

- W is the worklist of CFG nodes being processed. In each iteration of this algorithm, W is initialized to the set of nodes that contain assignments to V . Each node X in the worklist ensures that each node Y in $DF(X)$ receives a ϕ -function. Each iteration terminates when the worklist becomes empty.
- $HasAlready$ is an array of flags, one for each node, where $HasAlready(X)$ indicates whether a ϕ -function for V has already been inserted at X .

```
for each variable V
    HasAlready  $\leftarrow \emptyset$ 
    WorkList  $\leftarrow \emptyset$ 
    for each node X that may modify V
        WorkList  $\leftarrow$  WorkList  $\cup$   $\{X\}$ 
    while WorkList  $\neq \emptyset$ 
        remove X from W
        for each Y  $\in$   $DF(X)$ 
            if Y  $\notin$  HasAlready then
                insert a  $\phi$ -node for V at Y
                HasAlready  $\leftarrow$  HasAlready  $\cup$   $\{Y\}$ 
            if Y  $\notin$  WorkList then
                WorkList  $\leftarrow$  WorkList  $\cup$   $\{Y\}$ 
```

Figure 3.4: Algorithm to insert ϕ -functions

3.3.4 Renaming of variables

The algorithm in figure 3.5 renames all mentions of variables. New variables denoted by V_i , where i is an integer incremented by 1 after every new definition of V in the program, are generated for each variable V . The visit to a node processes the

statements associated with the node in sequential order, starting with any ϕ -functions that may have been inserted. The data structures used are:

- *Stacks* is an array of stacks, one stack for each original variable V . The stacks can hold integers. The integer i at the top of $S(V)$ is used to construct the variable V_i that should replace a use of V .
- *Counters* is an array of counters, one for each original variable V . The counter value $Counters[V]$ tell how many assignments to V have been processed.

procedure *Rename*(Block X)

for each ϕ -node P in X

GenName(LHS(P))

for each statement A in X

for each variable $V \in RHS(A)$

 replace V by V_i , where $i = Top(Stacks[V])$

for each variable $V \in LHS(A)$

GenName(V)

for each $Y \in Succ(X)$

$j \leftarrow$ position in Y 's ϕ -functions corresponding to X

for each ϕ -node P in Y

 replace the j th operand of RHS(P) by V_i

 where $i = Top(Stacks[V])$

for each $Y \in Children(X)$

Rename(Y)

for each ϕ -node or statement A in X

for each $V_i \in LHS(A)$

 pop $Stacks[V]$

procedure *GenName*(Variable V)

$i \leftarrow Counters[V]$

 replace V by V_i

 push i onto $Stacks[V]$

$Counters[V] \leftarrow i + 1$

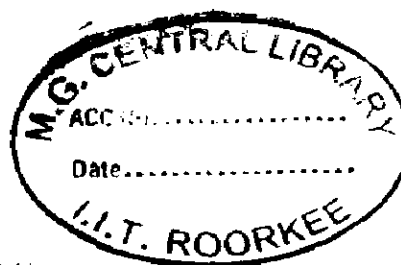


Figure 3.5: Algorithm for renaming of variables

The program to convert simple C language program into SSA is written. The program is divided into two modules. In first module, using Lex and YACC, the parser is designed for CFG file. In this, CFG for input program is generated and dominance frontier set is constructed from this CFG and also the position of ϕ -functions for each variable is computed. In first module, the CFG file is parsed using grammar given in section 3.2.2. The required structures are constructed by using algorithms given in sections 3.3.2 and 3.3.3. Each block in CFG file is parsed and the structures are computed.

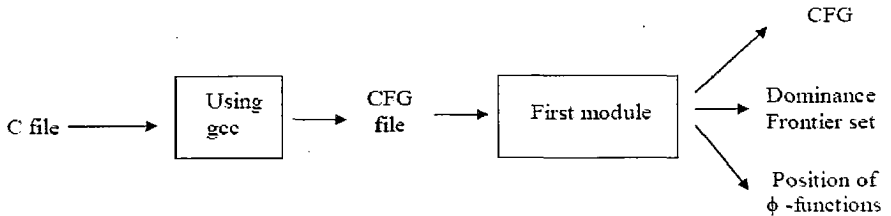


Figure 3.6 : Structures computed after first module

In second module, using Lex and YACC, the input CFG file is parsed again to write actual SSA code in output file. The variables are renamed as per the algorithm given in section 3.3.4 while writing into output file. Each block in CFG is parsed and written in output file as it appears in the program after renaming of variables and ϕ -functions are inserted in the blocks. Some data structures are passed from first module to second module using named pipes (fifos).

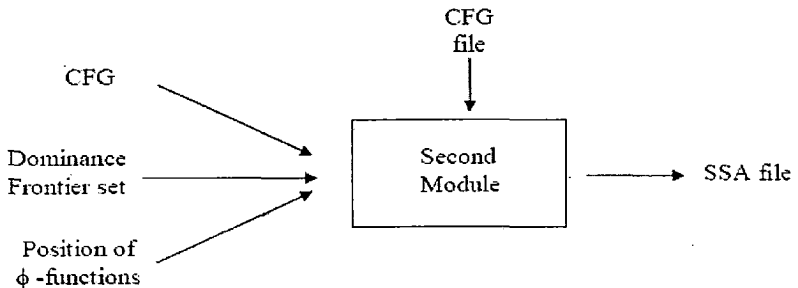


Figure 3.7 : Output after second module

3.3.5 CFG Specification File

Consider input C language program is test.c. So the CFG specification file is generated using command 'gcc -fdump-tree-cfg'. CFG generated from CFG specification file for C language program as shown in figure 3.8. Entry block is a starting block through which program starts execution. Exit block is an ending block through which program exits. Program control jumps from one block to another block as per the arrow directions.

```

int main()
{
    int i, j, k, l;
    i=j=k=l=1;
    do
    {
        if(i==1)
        {
            j=j;
            if(l==1)
                l=2;
            else
                l=3;
            k=k+1;
        }
        else
            k=k+2;
        printf("%d %d %d %d\n", i, j, k, l);
        do
        {
            if(k==1)
                l=1+4;
        }while(l==3);
        i=i+6;
    }while(i==7);
}

```

- (2)
- (3)
- (3)
- (3)
- (4)
- (4)
- (5)
- (6)
- (7)
- (8)
- (9)
- (10)
- (11)
- (12)
- (13)
- (13)

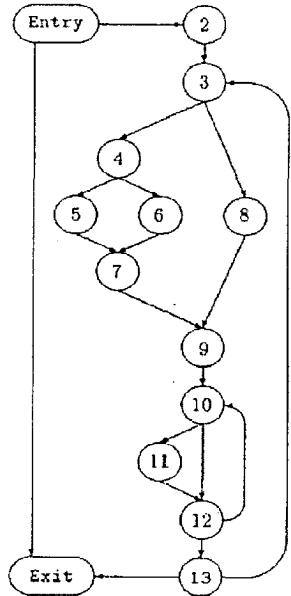


Figure 3.8 : C language program and its Control Flow Graph

3.3.6 Construction of Dominance frontier sets

After constructing CFG from CFG specification file, the dominance frontiers set for each node are constructed by parsing the cfg specification file by first module of project.

The name of object file created is 'dominance'. So dominance frontier sets are constructed by executing 'dominance test.cfg'. Here, test.cfg is CFG specification file

generated by gcc. Also, in first module, the positions of ϕ -functions inserted at each node are computed. The dominance frontier sets for each node after executing first module are as:

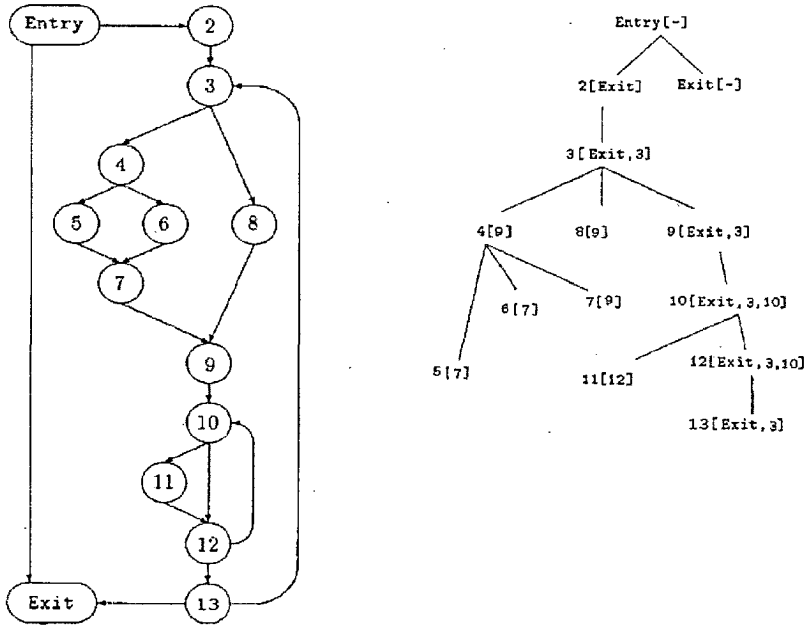


Figure 3.9 : CFG and dominance frontier set for each node

As shown in figure 3.9, the block number in bracket indicates dominate frontier for the node.

3.3.7 Writing output in the file

In second module, the CFG specification file is parsed again and the SSA for input program is actually written in the output file. The CFG specification file is given as input to the second module. Some of the important structures such as CFG, dominance frontier set, position of ϕ -functions computed in first module are passed as input to second module by using named pipes(ffo). The object file of second module

is 'rename'. So after executing command ' rename test.cfg', the output is written into the file 'ssa.txt' shown as:

Output file : ssa.txt

```
main ()
{
    int l;
    int k;
    int j;
    int i;
    <bb 2>:
    i_1 = 1 ;
    j_1 = 1 ;
    k_1 = 1 ;
    l_1 = 1 ;
    <bb 3>:
    j_2 = PHI(j_1, j_4) ;
    l_2 = PHI(l_1, l_8) ;
    k_2 = PHI(k_1, k_4) ;
    i_2 = PHI(i_1, i_3) ;
    if ( i_2 == 1 )
        goto <bb 4>;
    else
        goto <bb 8>;
    <bb 4>:
    j_3 = i_2 ;
    if ( l_2 == 1 )
        goto <bb 5>;
    else
        goto <bb 6>;
    <bb 5>:
    l_3 = 2 ;
    goto <bb 7>;
    <bb 6>:
    l_9 = 3 ;
    <bb 7>:
    l_4 = PHI(l_3, l_9) ;
    k_3 = k_2 + 1 ;
    goto <bb 9>;
    <bb 8>:
    k_5 = k_2 + 2 ;
    <bb 9>:
    j_4 = PHI(j_3, j_2) ;
    l_5 = PHI(l_4, l_2) ;
```

```

    k_4 = PHI(k_3, k_5) ;
    printf(&"%d %d %d %d\n"[0], i_2, j_4, k_4,
    l_5)
<bb 10>:
    l_6 = PHI(l_5, l_8) ;
    if ( k_4 == 1 )
        goto <bb 11>;
    else
        goto <bb 12>;
<bb 11>:
    l_7 = l_6 + 4 ;
<bb 12>:
    l_8 = PHI(l_6, l_7) ;
    if ( l_8 == 3 )
        goto <bb 10>;
    else
        goto <bb 13>;
<bb 13>:
    i_3 = i_2 + 6 ;
    if ( i_3 == 7 )
        goto <bb 3>;
    else
        goto <bb 14>;
<bb 14>:
    return;
}

```

3.4 Applying Constant Propagation Algorithm

After converting C code into SSA, the program to apply constant propagation algorithm on this SSA code is developed. In this program alias classes for pointer variables are also constructed and the algorithm is applied for these variables also.

Consider the simple test1 program in C language as shown in figure 3.10. This test program contains constants and pointers. There is also SSA form of this program obtained after applying SSA construction algorithm on test1 program.

```

int main()
{
    int i, j, k, l, m;
    int *p = &j, *q = &k;
    i = 1;
    j = 1;
    k = 1;
    l = 1;
    scanf("%d", &m);
    if( i == 1)
    {
        if( l == 1)
        {
            *p = 7;
            l = 2;
            m = l + j;
            i = k + j;
        }
        else
        {
            *q = 8;
            l = 3;
            m = l + k;
        }
    }
}

```

```

main ()
{
    int *q, *p, m, l, k, j i;
    int m.5, k.4, j.3, k.2, m.1, j.0;
<bb 2>:
    p = &j ;
    q = &k ;
    i_1 = 1 ;
    j = 1 ;
    k = 1 ;
    l_1 = 1 ;
    scanf(s"%d", &m)
    if ( i_1 == 1 )
        goto <bb 3>;
    else
        goto <bb 6>;
<bb 3>:
    l_2 = PHI(l_1, l_8) ;
    i_2 = PHI(i_1, i_3) ;
    if ( l_2 == 1 )
        goto <bb 4>;
    else
        goto <bb 5>;
<bb 4>:
    *p = 7 ;
    l_3 = 2 ;
    j.0 = j ;
    m.1 = l_3 + j.0 ;
    m = m.1 ;
    k.2 = k ;
    j.3 = j ;
    i_3 = k.2 + j.3 ;
    goto <bb 6>;
<bb 5>:
    *q = 8 ;
    l_3 = 3 ;
    k.4 = k ;
    m.5 = l_3 + k.4 ;
    m = m.5 ;
<bb 6>:
    return;
}

```

Figure 3.10 : Test1 program and its SSA form

Then constant propagation algorithm is applied. After applying constant propagation algorithm with alias classes for pointers, the output is as shown at right hand side of figure 3.11.

```

main ()
{
    int *q, *p, m, l, k, j, i;
    int m.5, k.4, j.3, k.2, m.1, j.0;
<bb 2>:
    p = *j ;
    q = *k ;
    i_1 = 1 ;
    j = 1 ;
    k = 1 ;
    l_1 = 1 ;
    scanf("%sd"[0], &m)
    if ( i_1 = 1 )
        goto <bb 3>;
    else
        goto <bb 6>;
<bb 3>:
    l_2 = PHI(l_1, l_8) ;
    i_2 = PHI(i_1, i_3) ;
    if ( l_2 = 1 )
        goto <bb 4>;
    else
        goto <bb 5>;
<bb 4>:
    *p = 7 ;
    l_3 = 2 ;
    j.0 = j ;
    m.1 = l_3 + j.0 ;
    m = m.1 ;
    k.2 = k ;
    j.3 = j ;
    i_3 = k.2 + j.3 ;
    goto <bb 6>;
<bb 5>:
    *q = 8 ;
    l_3 = 3 ;
    k.4 = k ;
    m.5 = l_3 + k.4 ;
    m = m.5 ;
<bb 6>:
    return;
}

```

```

main ()
{
    int *q, *p, m, l, k, j, i;
    int m.5, k.4, j.3, k.2, m.1, j.0;
<bb 2>:
    p = *j ;
    q = *k ;
    scanf("%sd" [0], m );
    if ( 1 = 1 )
        goto <bb 3>;
    else
        goto <bb 6>;
<bb 3>:
    l_2 = PHI( 1, l_8 ) ;
    i_2 = PHI( 1, i_3 ) ;
    if ( l_2 = 1 )
        goto <bb 4>;
    else
        goto <bb 5>;
<bb 4>:
    j.0 = 7 ;
    m.1 = 2 + j.0 ;
    m = m.1 ;
    k.2 = 1 ;
    j.3 = 7 ;
    i_3 = k.2 + j.3 ;
    goto <bb 6>;
<bb 5>:
    k.4 = 8 ;
    m.5 = 2 + k.4 ;
    m = m.5 ;
<bb 6>:
    return;
}

```

Figure 3.11 : SSA code and optimization of SSA code

The new variables in this program are generated by compiler while generating three address code in intermediate representation. Above program contains much less instructions compare to original program in SSA form. If we apply constant propagation algorithm again(second pass), the program will be as:


```

main ()
{
    int *q, *p, m, l, k, j i;
    int m.5, k.4, j.3, k.2, m.1, j.0;
<bb 2>:
    p = &j ;
    q = &k ;
    scanf( "%d" [0], m );
    if ( l == 1 )
        goto <bb 3>;
    else
        goto <bb 6>;
<bb 3>:
    l_2 = PHI( l, l_8 ) ;
    i_2 = PHI( l, i_3 ) ;
    if ( l_2 == 1 )
        goto <bb 4>;
    else
        goto <bb 5>;
<bb 4>:
    m.1 = 2 + 7 ;
    m = m.1 ;
    i_3 = 1 + 7 ;
    goto <bb 6>;
<bb 5>:
    m.5 = 2 + 8 ;
    m = m.5 ;
<bb 6>:
    return;
}

```

In second pass, the program in SSA can be further optimized as seen from above code.

Chapter 4

RESULTS

4.1 Results after Applying Constant Propagation Algorithm

After converting input C program into SSA, constant propagation algorithm on some special test programs is applied. Test programs are designed in C language such that they contain more pointer variables.

Table 4.1 Comparison of applying constant propagation algorithm on SSA

SSA Program	SSA Code Instructions	Algorithm Applied	No of Instructions after applying Algorithm	% Improvement
Test1	55	With SSA without Alias classes	51	7.2
		With SSA and with Alias classes	47	14.4
Test2	70	With SSA without Alias classes	66	5.7
		With SSA and with Alias classes	60	14.2
Test3	100	With SSA without Alias classes	93	7.0
		With SSA and with Alias classes	85	15.0

Test1 program is already discussed in previous section. Only integers and pointers pointing to integers are used in test programs to avoid complexity while applying optimizing algorithm. Also arrays, structures and procedures are not used. First test programs are converted into intermediate code using GNU C Compiler (GCC).

4.2 Comparison

Figure 4.1 shows the number of instructions for three test programs before and after applying constant propagation algorithm. Series "Test Program" at top corresponds to number of instructions in original SSA code. Series "without Alias Classes" in middle shows number of instructions after applying constant propagation algorithm on SSA code without alias classes. Series "with Alias Classes" corresponds to applying the algorithm with alias classes. Applying the algorithm on SSA with alias classes yield good results for test programs.

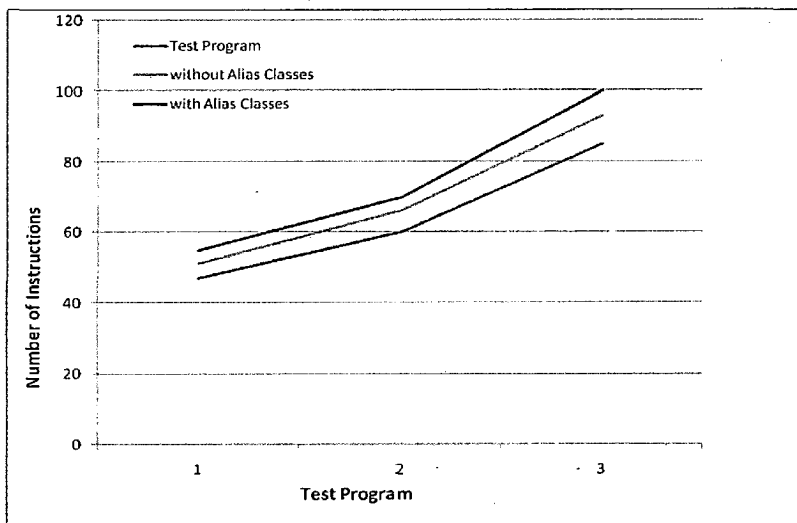


Figure 4.1: Number of instructions for optimized code

Figure 4.2 further shows improvement in the SSA code for three test programs. Series “without Alias Classes” shows % improvement in SSA code after applying constant propagation algorithm without using alias classes for pointers. Series “with Alias Classes” corresponds to applying the algorithm without using alias classes.

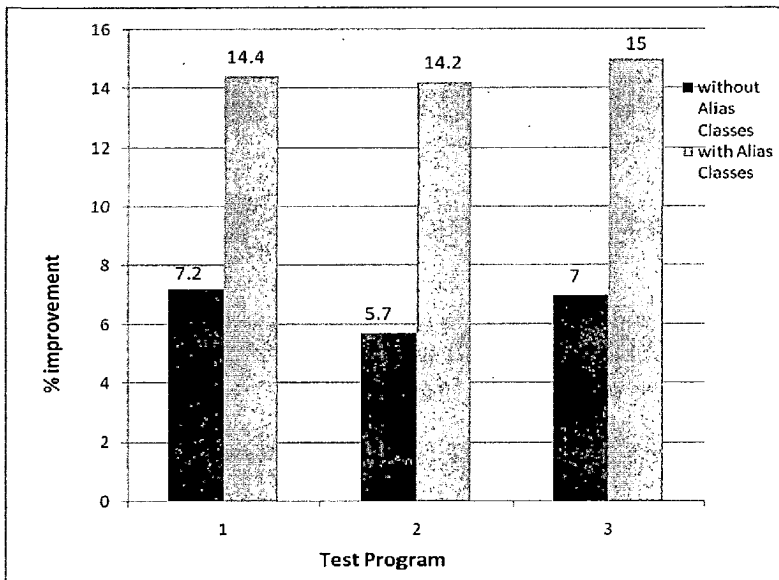


Figure 4.2 % improvement for various approaches

Chapter 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

In this dissertation, we have proposed a new approach to apply optimization algorithm on pointer variables using static single assignment based intermediate representation with alias classes for pointers. Using SSA, data flow analysis is not required and using alias classes, pointers can be optimized. Results show that applying SSA with alias classes produces more optimized code than applying either on original IR or on SSA without alias classes for the programs which contain large number of constants and pointer variables.

Though constant propagation algorithm applying on SSA with alias classes takes more time compared to SSA without alias classes, it produces efficient code. Also it improves the code twice as compared to applying the algorithm on SSA without alias classes.

5.2 Future Work

In GCC(Gnu Compiler Collection), SSA is implemented for all variables except pointers. In future, GCC code can be modified to implement SSA with alias classes. Also in this dissertation, only constant propagation algorithm is applied on SSA code with alias classes. Other optimization algorithms like global value numbering, copy propagation can also be applied. Further SSA itself can be improved to reduce the number of ϕ -functions in code.

REFERENCES

- [1] V. Alfred, R. Sethi, and D. U. Jeffrey, *Compilers: Principles, Techniques and Tools*: Addison-wesley, 1986.
- [2] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, pp. 451-490, 1991.
- [3] V. Sreedhar, R. Ju, D. Gillies, and V. Santhanam, "Translating out of static single assignment form," *In Proceedings of the Static Analysis Symposium, Lecture Notes in Computer Science*, vol. 1694, pp. 194-210, 1999.
- [4] S. S. Muchnick, *Advanced compiler design and implementation*: Morgan Kaufmann, 1997.
- [5] J. Aycock and N. Horspool, "Simple generation of static single-assignment form," *9th International Conference in Compiler Construction, Lecture Notes in Computer Science*, vol. 1781, pp. 110-125, 2000.
- [6] W. Amme and E. Zehendner, "Efficient calculation of data dependences in programs with pointers and structures," in *Proceedings of the 23rd EUROMICRO Conference, EUROMICRO 97. New Frontiers of Information Technology*, pp. 55-62, 2002.
- [7] B. B. S. Hack, D. Grund, F. Rastello, B. D. de Dinechin, and E. N. S. L. Stmicroelectronics, "Fast Liveness Checking for SSA-Form Programs," *In Proceedings of the Sixth Annual IEEE/ACM International Symposium on Code Generation and Optimization*, vol. 37, pp. 18-36, 2003.
- [8] D. Novillo, R. Unrau, and J. Schaeffer, "Concurrent ssa form in the presence of mutual exclusion," *In Proceedings of the International Conference on Parallel Processing*, pp. 356-364, 2002.
- [9] L. Séméria and G. De Micheli, "Resolution, optimization, and encoding of pointer variables for the behavioral synthesis from c," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, pp. 213-233, 2002.
- [10] D. R. Chase, M. Wegman, and F. K. Zadeck, "Analysis of pointers and structures," *ACM SIGPLAN Notices*, vol. 25, pp. 296-310, 1990.

- [11] G. Bilardi and K. Pingali, "Algorithms for computing the static single assignment form," *Journal of the ACM (JACM)*, vol. 50, pp. 375-425, 2003.
- [12] F. Rastello, F. De Ferrière, and C. Guillon, "Optimizing the translation out-of-SSA with renaming constraints," *In Proceedings of the International Symposium on Code Generation and Optimization*, pp. 265-278, 2005.
- [13] A. Lenart, C. Sadler, and S. K. S. Gupta, "SSA-based flow-sensitive type analysis: combining constant and type propagation," *In Proceedings of the ACM Symposium on Applied Computing*, pp. 813-817, 2000.
- [14] M. Sassa, Y. Ito, and M. Kohana, "Comparison and evaluation of back-translation algorithms for static single assignment forms," *Computer Languages, Systems & Structures*, vol. 35, pp. 173-195, 2009.
- [15] J. Von Ronne, N. Wang, and M. Franz, "Interpreting programs in static single assignment form," *In Proceedings of the ACM SIGPLAN 2004 Workshop on Interpreters, Virtual Machines and Emulators*, pp. 23-30, 2004.
- [16] S. Staiger, G. Vogel, S. Keul, E. Wiebe, "Interprocedural Static Single Assignment Form", *In Proceedings of the 14th Working Conference on Reverse Engineering*, pp. 1-10, 2007.
- [17] D. Novillo, "TreeSSA a new optimization infrastructure for GCC", *In Proceedings of the 2003 GCC Developers' Summit*, pp. 181-193, 2003.
- [18] S. Hack, D. Grund, and G. Goos, "Register allocation for programs in SSA-form", *In 15th International Conference on Compiler Construction*, volume 3923 of *Lecture Notes in Computer Science*, Springer, pp. 247-262, 2006.

LIST OF PUBLICATIONS

- [1] Deodatta Barhate, A. K. Sarje. "An approach for pointer optimization using SSA based intermediate representation," in *Proceedings of International Conference on Recent Trends in Information Technology*, pp. 398 – 401, June 3-5, 2011.