M.Tech. Dissertation

SSA based Partial Redundancy Elimination

Submitted in partial fulfillment of the requirements for the degree of
Master of Technology

By

Ashish Kundu
Roll No – 98305017

under the guidance of
Prof. D. M. Dhamdhere

Department of Computer Science and Engineering
Indian Institute of Technology
Bombay

January 21, 2000

Dissertation Approval Sheet

This is to certify that the dissertation entitled *SSA based Partial Redundancy Elimination* by **Ashish Kundu** is approved for the award of the degree of **Master of Technology**.

Guide :

Chairman :

Internal Examiner :

External Examiner :

Date :

Acknowledgement

I express my sincere gratitude towards my guide **Prof. D. M. Dhamdhere** for his constant help, encouragement and inspiration throughout the project work. Without his invaluable guidance, this work would never have been a complete and a successful one. Working under him was very challenging and enjoying. I would also like to thank **Prof. A. Sanyal** for his keen interest and help in this project.

Finally, I express my love and respect to my parents, brother and sister for their love, support, sacrifice and inspiration in each and every step of my life till today.

IIT, Bombay **Ashish Kundu**
January 11, 2000

Abstract

This thesis presents SSA based frameworks for Partial Redundancy Elimination (PRE) (called as E-path_PRE) and Strength Reduction of Large Expressions (SRLE) (called as E-path_SRLE) based on the notion of eliminatability paths. Compared to earlier PRE work, the advantages of our approach are correctness, efficiency and simplicity. Our approach uses only well-known fundamental data flows of available expressions and anticipatable (*i.e.* very-busy) expressions. The algorithms are of linear time complexity. They incur less cost and insert a smaller number of computations as compared to earlier approaches. We have also remedied the correctness problems that some of the earlier algorithms suffer from.

In this thesis, we have presented our SSA based algorithms for computation of availability and anticipatability. The algorithms of E-path_PRE and E-path_SRLE have also been explained and have been compared with the earlier solutions of PRE and SRLE. Correctness proofs have been presented, wherever necessary. The E-path_PRE prototype developed by us has been described along with some result.

Contents

List of Figures

Chapter 1

Introduction

1.1 Aim of the Project

The aim of this project is to formulate partial redundancy elimination (PRE) and strength reduction of large expressions (SRLE) in SSA form. Given an input program which is already in SSA form, our optimization technique can be used to eliminate partial redundancies and strength reduce the expressions having at least one high strength operator. The output is either a PRE-optimized program or an SRLE-optimized program in SSA form.

1.2 Motivation

Code optimization in compilers is the process of improving the code through static analysis (known as data flow analysis) of the program. The improvement may be on the basis of code size, runtime memory requirement or the amount of computations or all of these. Depending on the optimization to be performed, the data flow analysis can be unidirectional or bidirectional [KD94, MR79]. Computation of availability, anticipatability, live variable analysis etc. are some examples of unidirectional data flow problems. Partial redundancy elimination [MR79, KD94, DP93], composite hoisting and strength reduction [JD82, Dha89] are some classical examples of bidirectional dataflow problem. Though solving unidirectional data flow problems is simpler and incurs less cost than solving bidirectional dataflow problems, bidirectional dataflow problems are more powerful. The optimization technique of partial redundancy elimination (PRE) is very powerful as it subsumes classical optimizations like code movement, loop invariant movement and common subexpression elimination. An example of PRE is given in figure 1.1. In figure 1.1(a), expression $b + c$ is partially

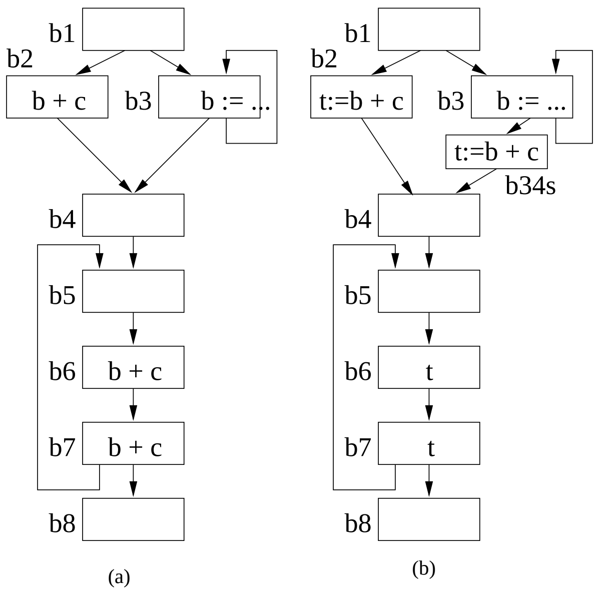

Figure 1.1: (a) is the original PFG. (b) is the PFG obtained after partial redundancy elimination.

redundant along the path $[b_2, b_4, b_5, b_6]$, $[b_6, b_7]$[1] and $[b_7, b_5, b_6]$. Also $b + c$ is a loop invariant for the loop with b_5 as loop-entry. The program after elimination of the above partially redundancies is shown in figure 1.1(b). The expression is not available along the path $[b_3, b_4]$. Therefore the insertion of the expression is made along the edge (b_3, b_4). The partially redundant expression in b_6 and b_7 have been replaced by the uses of the temporary of $b + c$. b_{34s} is the synthetic node inserted along edge (b_3, b_4).

Another machine-independent powerful code optimization technique is strength reduction of large expressions (SRLE) [DD95]. It subsumes PRE and strength reduction. In addition, SRLE-optimized code is more optimal than the code optimized by standard strength reduction techniques. The figures 1.2, 1.3 and 1.4 present an example of SRLE. In figure 1.2, the expression $i * k + j * l$ can be strength reduced as j is an induction variable and i, k and l are loop constants. The expression has been written in three-address form. After application of conventional strength reduction technique (see figure 1.3), the expression $j * l$ is strength reduced and since $i * k$ is a loop invariant it gets hoisted to the node b_1. The figure 1.4 is the resulting program after application of the strength of large expression. In figure 1.3 node b_5 contains an addition $T_3 = T_1 + T_2$. But SRLE performs this

[1]Actually along $[b_6, b_7]$ $b + c$ is totally redundant. All totally redundant computation are also partially redundant computation.

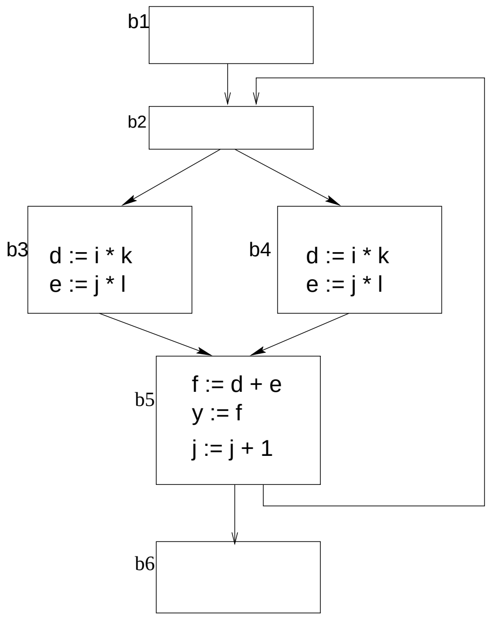

Figure 1.2: Original PFG for SRLE.

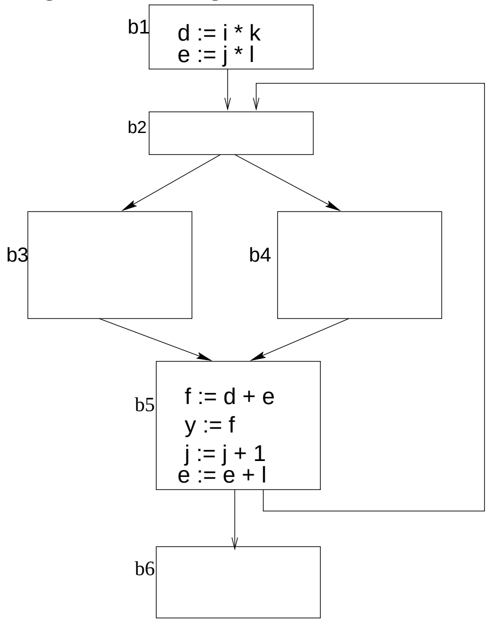

Figure 1.3: PFG after standard strength reduction.

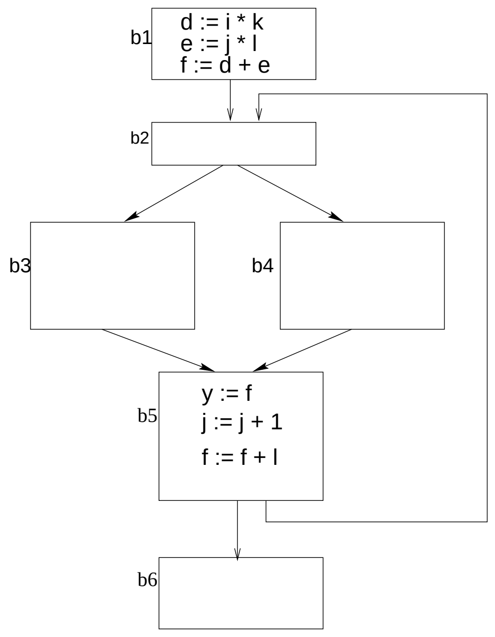

Figure 1.4: PFG after standard strength reduction of large expression.

addition before the loop entry in node b_1 and thereby producing better optimization. Most of the commercial compilers do not implement PRE and SRLE due to the quadratic time complexity and complex data flows involved in the bit vector frameworks.

Intermediate representations coupled with the framework used in compilers for data flow analysis affect the efficiency of optimization significantly. Intermediate representations are used by compilers to hide the processor architecture during optimizations. Bit vector framework is commonly used in traditional compilers for data flow analysis. In this framework the minimum number of passes on the program needed for computing the maximal fixed point (MFP) of the transformation is nonlinear in terms of the number of basic blocks in the corresponding program flow graph. Static single assignment (SSA) is a sparse-graph program representation used in modern optimizing compilers [C^+91], whose appeal stems from the concise representation of def-use information. The sparse-ness property of the SSA form encourages its use in data flow analysis because it leads to considerable reduction in the cost of analysis. Also SSA makes the data flow analysis insensitive to the back edges. The reason is — in the SSA form of a program a variable has exactly one definition and all its uses are dominated by the definition. An example in figure 1.5 is used to illustrate the SSA form. Figure 1.5(a) is the original program to be transformed to SSA form. In figure 1.5(b) the original variables a, b and c have been given versions. Note that each

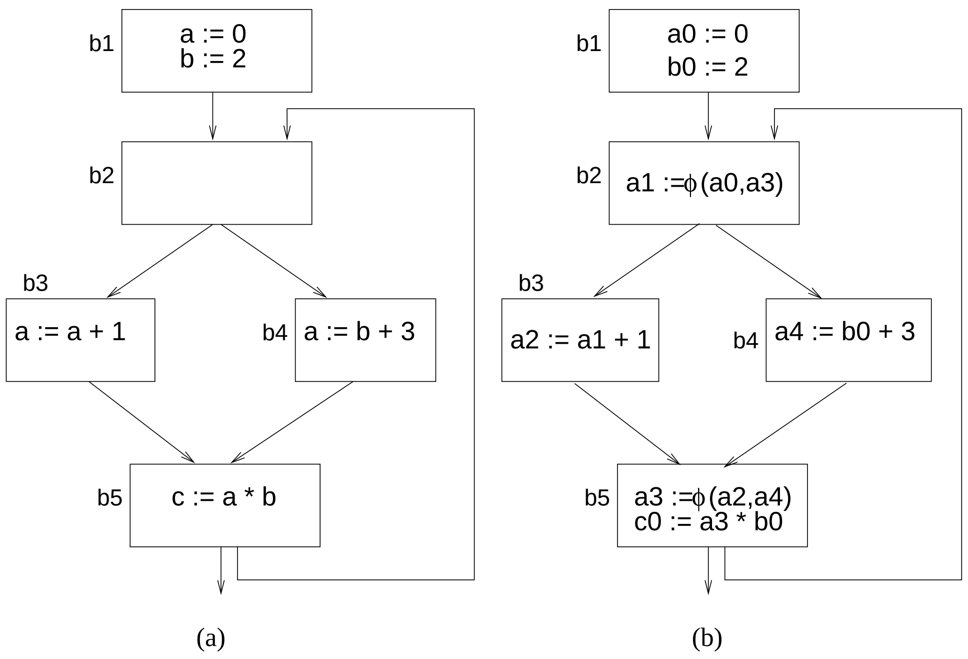

Figure 1.5: (a) Original PFG. (b) PFG in SSA form.

uniquely versioned variable has a single assignment. In the join node b_2 and b_5, ϕ-functions for a have been inserted to specify that more than one value of a are reaching at that point.

[DRZ92] had addressed the issue of using the SSA form for PRE optimizations. Chow *et al* [C$^+$97] have proposed a PRE algorithm based on the SSA form. They use the lazy code motion approach of [KRS92] and cast it in an SSA-based algorithm for PRE. However, we have detected that their algorithm suffers from correctness issue. In addition, it is very complex and has a few deficiencies discussed later in chapter 2.

1.3 Terminology

In this section we are defining some terminology, which have been used later in this report. Some of them are defined in [DD95] (we are reproducing them as we have used those concepts). At the beginning the terminologies common to both PRE and SRLE have been included. In the next section, terminologies specific to SRLE have been given.

1.3.1 Terminologies common to PRE and SRLE

Empty Node [DD95] : A node is empty with respect to an expression e, iff it does not contain any occurrence of e nor any assignment to any operand of e.

Locally Anticipatable [MR79] : An expression e is locally anticipatable in a block if and only if, there is at least one occurrence of e in the block and there is no change in any of the operands of e before the first occurrence in the block.

Locally Available [DD95] : An expression e is locally available at the exit of a block if and only if, there is at least one occurrence of e in the block and there is no change in any of the operands of e after the last occurrence in the block.

Anticipatable [MR79] : An expression e is anticipatable at a point p in the program iff along each path from p to the exit of the program there is an occurrence of e and no operand of e gets changed between p and the first occurrence along the path.

Partially Anticipatable [MR79] : An expression e is partially anticipatable at a point p in the program, iff there is at least one path from p to the exit of the program, along which there is an occurrence of e before any of its operands gets changed.

Available [MR79] : An expression e is available at a point p in a program iff along each path from entry of the program to p, there is an occurrence of e and no operand of e gets changed between this occurrence and p.

Partially Available [MR79, Dha90a] : An expression e is partially available at a point p in a program iff along at least one path from entry of the program to p, there is an occurrence of e and no operand of e gets changed between this occurrence and p.

Partially Redundant Occurrence [MR79] : An occurrence of an expression e at a point p in the program is partially redundant iff the expression is partially available at p.

Available-at-exit : An expression is available-at-exit in a node iff it is available at the exit of the node.

Available-at-exit Occurrence : An expression e has an available-at-exit occurrence in a node iff x contains an occurrence of e which is available at the exit of the node.

Safe [DD95, KD] : An expression e is safe at a point in the program iff e is anticipatable or available at that point.

Eliminatable Occurrence [DD95, KD] : An occurrence of an expression e in a node k is an eliminatable occurrence, if the occurrence is locally anticipatable in k and there exists a path $[i \dots k]$ such that :

- e is locally available at the exit of i;
- all nodes on the path $(i \dots k)$ are empty with respect to e and
- e is safe at the exit of each node on the path $[i \dots k)$.

Such a path $[i \dots k]$ is called an *eliminatable (or equivalently eliminatability) path* with respect to e. The terms *eliminatability path* and *e-path* have been used interchangeably in this report.

Large Expression [DD95] : An expression e is a large expression iff it involves more than one high strength operator(s) like multiplication operator or exponentiation operator.

Loop Path : A loop path is a path $[i \dots k]$ in a program iff i is a loop entry and k is the loop exit (of the same loop) and all the nodes in $[i \dots k]$ are inside the loop body such that a path is possible among them. If all the nodes in $[i \dots k]$ are empty with respect to an expression e, then we call such a path as *empty loop path* with respect to e.

Splitted Node : If a node x in a program flow graph contains occurrence(s) of an expression(s) such that e is both locally anticipatable and locally available at exit in node x, then the node can be splitted into two nodes x_{in} and x_{out} such that e is only locally anticipatable in x_{in} and locally available at exit of x_{out}. We call x_{in} as *innode* and x_{out} as *outnode*. Moreover all incoming edges of x go to x_{in} and all outgoing edges of x come out from x_{out}. x_{in} does not have any outgoing edge and x_{out} does not have any incoming edges.

Maximal Path : A path p is a maximal path in a graph G, iff no more node of G can be added to p, unless p is a cyclic path. Path p involves a cycle and p is a maximal path, iff only one iteration of the cyclic path is present in p.

Eliminatability Graph [KD] : A graph G is an eliminatability graph with respect to an expression e, iff any maximal path in the graph is an eliminatable path with respect to e. The terms *egraph* (or e-graph) and *eliminatability graph* have been used interchangeably in this report.

Extraneous Φ-function [C+97] : A Φ-function for an expression e in a node x is an extraneous Φ-function iff either it has no use in the program or

- e is not available at the entry of x; and
- if $[i \ldots k]$ is an eliminatable path for e in the program flow graph, then x must not be a node in $(i \ldots k]$.

Useless Node [KD] : A node x is a useless node with respect to an expression e, iff it cannot be a part of any eliminatable paths for e due to safety reasons, or no successor and/or predecessor of x is involved in an eliminatable path or x is neither empty nor contains a locally anticipatable occurrence nor contains an available at the exit occurrence.

Critical Edge [DP93] : An edge (b_1, b_2) is told to be a critical edge, iff b_1 has at least one more outgoing edge in the program flow graph except this edge and b_2 has at least one more incoming edge.

1.3.2 Terminologies specific to SRLE

Update definition : A definition of the form $v = v \pm c$ is a update definition.

Redefinition : A definition of any form other than a update definition is a Redefinition.

Update computation : Any computation e_i of type $T = T \pm C$, where T is a variable and C is a region constant is called an update computation.

Recomputation : A computation for e_i which is of any type other than a update computation is a recomputation.

X-anticipatability : An expression e is X-anticipatable at a point p in basic block b in the program, if each path emanating from the exit of b has an occurrence of e and no operand of e is defined by a Redefinition between p and the point of occurrence of e along the paths.

srle-availability : An expression e is srle-available at a program point p, iff along all paths from entry to p in the program, there occurs a computation of e followed by exactly one update definition for an operand of e and no redefinition.

srle-anticipatability : An expression e is srle-anticipatable at a program point p, iff along all paths from p to exit in the program, there occurs a computation of e preceded by exactly one update definition for an operand of e and redefinition.

Xsrle-local-availability (Xsrle-avail) : An expression e is Xsrle-avail at the exit of a node x iff there exists at least one update definition of an operand of e following the last occurrence of e in x and no redefinition.

Xsrle-local-anticipatability (Xsrle-antloc) : An expression e is Xsrle-antloc at the entry of a node x iff there exists at least one update definition of an operand of e preceding the first occurrence of e and no redefinition.

Xsrle-both - A node is said to be of type Xsrle-both with respect to e iff it is of type either

- Xsrle-antloc and avail or
- antloc and Xsrle-avail or
- Xsrle-antloc and Xsrle-avail with respect to e.

Xsrle-empty : A node is said to be Xsrle-empty with respect to e iff it contains at least one update definition of an operand of e and no redefinition or occurrence with respect to e.

srle-safety at exit of x : An expression e is said to be srle-safe at exit p of node x in the program iff

- e is srle-available AND srle-anticipatable at p.

srle-eliminatability path (srle-e-path) : A path $[i \dots j \dots k]$ is an srle-eliminatability path with respect to an expression e iff

- i contains an avail or Xsrle-avail occurrence of e,
- k contains an antloc or Xsrle-antloc occurrence of e,
- j is empty or Xsrle-empty
- j satisfies safety or srle-safety.

The srle-counterpart of egraph is called as srle-egraph.

The global sr-property is the logical OR of the corresponding normal property and srle-property. So sr-availability is availability or srle-availability. The local sr-property is the logical OR of the corresponding normal property and Xsrle-property.

SRLE-candidate expression : Such an expression e has occurrences in a loop and involves high strength operators like $*$ and $**$ (exponentiation operator).

Proper SRLE-candidate expression : Such an expression e has occurrences in a loop and involves high strength operators like $*$ and $**$ (exponentiation operator) subject to the condition that subexpressions involving $*$ or $**$ are of type $a * i$ or $a * *i$ respectively where i is an induction variable a is a loop constant in the loop concerned.

Φ_1*-function* : In this type of Φ-function, the operand is a single unpaired-variable. Its structure is $\Phi(h, \dots, h)$. This identical to the normal Φ-function.

Φ_2*-function* - Φ_2 is the srle-counterpart of the Φ_1 (or Φ). This is of type

$$(h, s) \leftarrow \Phi_2((h, s), \dots, (h, s)).$$

Each ordered pair (h, s) is for a single predecessor. The h in this pair is the normal Φ-operand in PRESSA form. The s in the pair is for computing srle-properties. Therefore s is the srle-counterpart of h.

For convenience we shall be using def and definition of a variable in an interchangeable manner. Node and basic blocks, dominator tree and dom tree have also been used interchangeably.

1.4 Organization of the Thesis

The thesis has been organized as follows. Chapter 2 presents the literature survey on PRE, SSA and SRLE. Section 2.1 talks about the concepts of SSA and the computation of minimal SSA form using Cytron *et al*'s algorithm. In section 2.1.5, we have presented a comparison between bit vector based data flow analysis and SSA based data flow analysis. In the next section 2.2, the technique and algorithms of SSA based partial redundancy elimination [C$^+$97] have been discussed. SSAPRE has correctness problem with computing some properties. It has been discussed and illustrated in section 2.2.2. SSAPRE has been analyzed in section 2.2.3. The optimization technique of strength reduction of large expressions and the placement criteria are included in section 2.3 and 2.3.1 respectively. The strength reduction scheme of [K$^+$98] is a formulation of CHSA in SSA. Section 2.4 discusses this scheme.

Chapter 3 presents a brief overview of our approach to PRE and SRLE : E-path_PRE [KD] in section 3.1 and E-path_SRLE in section 3.2. In the next chapter, we have presented, illustrated with an example and analyzed the algorithms [KD] of E-path_PRE. Correctness proofs have also been provided, wherever necessary. Each of the sections between 4.1 to section 4.8 have been used for the algorithm of one stage of E-path_PRE. In chapter 5, technique for performing SRLE in an SSA framework is discussed. Since E-path_SRLE is an extension of E-path_PRE, we have given more stress to the computations of SRLESSA form along with srle and Xsrle-properties. The sections from 5.1 to 5.6 talk about the extension of stages of E-path_PRE. Chapter 6 analyses and compares E-path_PRE with SSAPRE and E-path_SRLE with SRLE of [DD95] and at the end it concludes the work done in this project. Appendix A has described the interface and salient features of the E-path_PRE prototype that we have implemented. In section A.3, output of the prototype for a sample program has been included.

Chapter 2

Literature Survey

The original formulation of PRE by Morel-Renvoise [MR79] involved complex bi-directional data flows and suffered from problems of redundant code movement and missed opportunities of optimization [Dha88a, DS88]. Many researchers have tried to improve the formulation of PRE to eliminate the deficiencies of MRA, simplify the data flows and reduce their solution complexity [Dha88a, DS88, DRZ92, KRS92, DK93, DS93, KD94, DD95, KD99]. The basic PRE framework has also been extended to include strength reduction optimization [JD82, Dha89, DD95, K^+98] and to use it for other applications like live range determination in register assignment [Dha88b, Dha90b, L^+98]. The first and most significant contribution on SRLE has been done by Dhaneshwar and Dhamdhere [DD95]. Their algorithm is bit-vector based and has quadratic time complexity in worst case.

Static Single Assignment form gained popularity after the publication of [C^+91]. Cytron *et al*'s simple and efficient technique for computation of minimal SSA form of a program has been the basis on which compiler researchers have started formulating different data flow problems. Recently the optimization frameworks in SSA for partial redundancy elimination [C^+97] and strength reduction [K^+98] have been developed. It has been detected by us that the PRE framework in SSA of [C^+97] does not compute some properties correctly. SRLE has also not been formulated SSA.

2.1 SSA-based Data flow Analysis

In this section Static Single Assignment (SSA) form has been presented in detail. Algorithms to compute SSA form of a program have also been discussed by taking an example. A comparative study between the SSA-based optimization and the bit vector based optimization is also presented.

2.1.1 Static Single Assignment Form

Static Single Assignment [C$^+$91] form of a program is a sparse representation. As mentioned in Chapter 1, sparse representation is concerned only with the program points where the value of a program entity changes or where the entity actually occurs in the program. This decreases the size of the memory required for the otherwise large bit-vectors as well as the processing cost of those vectors. The definition of SSA form is as under :

Definition : *A program is defined to be in SSA form, iff each variable is defined by only one assignment statement and every use of each variable is dominated by its definition.*

From the definition, the meaning of the term *single* in SSA is apparent — every variable in the program has exactly one assignment. By the term *static* in SSA it is meant that if a static scan of the program in SSA form is made, then exactly one definition of each variable will be detected. It is not called dynamic because during runtime, a variable can be assigned more than once if its assignment is in a loop.

If a variable is not defined before its use in the program then, a dummy initialization of the variable is introduced while computing the SSA form of the program. This is to ensure that the final SSA form of the program does not violate the definition of SSA form. If two or more non-null paths meet at a single node[1], then any of the earlier definitions of a variable may not dominate any of the uses of that variable in this join node or later in the program. In this case ϕ-functions are introduced at the entry of this join node. Most commonly ϕ-functions are useful only in intermediate stage to ensure that the optimization does not violate semantic correctness of the program. In fact ϕ-function is an abstract notion used at a point for a variable, only to denote that at that program point, two or more values of the variable may be reaching. By the property of SSA, the target of each ϕ-function is a new variable. It is worth-noting here that each new assignment of a variable a creates a new version of a (*e.g.* $a_1, a_2, \ldots$). By the use of versioning def-use chaining is implicitly built into the program. The number of the arguments[2] passed through any ϕ-function is equal to the number of the predecessors of the join node in question. There are a number of techniques developed to compute SSA form of a program [C$^+$91, BM, BCS].

[1]Such a node is called a join node

[2]Henceforth the arguments of ϕ-functions will be called as ϕ-operands.

2.1.2 Some Terminology related to SSA

Below we are presenting some terminology used in Cytron et al's algorithm for computation of SSA form and also later in the report.

- *Strict Dominance* : A node X strictly dominates Y iff X dominates Y and $X \neq Y$.

  From the above definition, a node X does not strictly dominate Y iff either

  - X dominates Y and $X = Y$ or
  - X does not dominate Y (in this case obviously $X \neq Y$)

- *Dominance Frontier* $(DF(X))$: Node $Y \in DF(X)$, (where $DF(X)$ is the dominance frontier of X), if

  - X dominates a predecessor of Y and
  - X does not strictly dominate Y.

- *Iterated Dominance Frontier* $(DF^+(X))$: This is the positive closure of dominance frontier.

  $DF^+(X) = DF(X) \cup \bigcup_{Y \in DF(X)} DF^+(Y)$

2.1.3 Computing Minimal SSA Form

A program is said to be in minimal SSA form if the program is in SSA form and it involves the optimal number of versions for each variable and optimal number of ϕ-functions without violating the definition of SSA form. Computing SSA form of a program involves finding the program points containing definitions of variables and the join nodes so that new versions are created and ϕ-functions can be inserted at proper places. Performing this using naive approach may not give minimal SSA form. Cytron *et al* [C$^+$91] have devised algorithms to compute SSA form of a program efficiently.

2.1.3.1 Algorithms

Input to the computation is a program flow graph. The output shall be another program flow graph in SSA form.

Let the input program flow graph be $G = \langle N, E, n_0 \rangle$. Dominator tree of G can be computed from standard algorithms [ASU86, Muc97]. From the dominator tree $DF(X)$ is computed for all nodes $X \in N$. Computing $DF(X)$ is done in two steps. For this purpose two parameters called DF_{local} and DF_{up} are also computed. Their definition follows.

$DF_{local}(X) = \{Y \in Succ(X) \mid X \textit{ does not strictly dominate } Y\}$

$DF_{up}(Y) = \{Z \in DF(Y) \mid idom(Y) \textit{ does not strictly dominate } Z\}$

where $Succ(X)$ is a set of successor of nodes of X in the program flow graph, $idom(Y)$ is the immediate dominator of Y in the dominator tree. The dominance frontier defined in terms of DF_{local} and DF_{up} is as follows.

$DF(X) = DF_{local}(X) \cup \bigcup_{Y \in Children(X)} DF_{up}(Y).$

where $Children(X)$ is the set of children nodes of X in the dominator tree.

This reduces the cost of computing dominance frontier to linear [C$^+$91] cost (linear in the number of nodes in G). For the algorithm computing $DF(X)$ in above manner, we refer the reader to [C$^+$91]. The two stages of computing SSA form are discussed in detail with explanations.

2.1.3.2 *ϕ-Insertion*

In this stage the ϕ-functions for a variable are inserted at the entry of the iterated dominance frontiers of nodes containing definition(s) of that variable.

The detailed and original algorithm for computing SSA form can be found in [C$^+$91]. This *ϕ-insertion* algorithm is a worklist based algorithm. Initially worklist W contains only those nodes having at least one assignment to any of the variables. In each iteration of the algorithm a node X is removed from W and checked if in any node $Y \in DF(X)$ a $\langle V \leftarrow \phi(V, \ldots, V) \rangle$ is inserted or not for the current variable in question. If no, then it inserts one and inserts Y in worklist W. This insertion of Y to W gives the effect of iterated dominance frontier. This is repeated until the worklist becomes empty. Since the worklist is a set, there is no problem of repeated occurrence of the same node, thereby making the algorithm a nonlinear one. The cost of the algorithm is linear in terms of the number of nodes in the program.

2.1.3.3 *Renaming*

The next stage is renaming of variables or simply attaching versions to the variables. Each assignment and the target of each ϕ-function gets a new version.

The detailed and original algorithm for computing SSA form can be found in [C$^+$91]. The Renaming algorithm takes the output of ϕ-insertion algorithm as its input and generates the minimal SSA form. This algorithm uses an array of stacks of integers; each stack for a single original variable V. The integer i at the top of the stack is the most recent version used to create variable V_i, that will replace the *use* of V. So the top of the stack contains the most recent version assigned to V. The counter used in the algorithm contains the number for V which will be the next version; in other words it contains the number of the assignment statements including ϕ-functions processed whose target is V.

For the purpose of renaming a preorder traversal of the dominator tree is done and each time a node is visited following operations are performed.

```
For each assignment statement $A$ in node $X$,
    for all variables $V$ used in $A$ (i.e. $V$ is in the rhs of $A$),
        Replace $V$ by $V_i$, where $i$ = top of variable stack for $V$.
    For each variable $V$, which is a target of $A$,
        Replace $V$ by $V_j$, where $j$ is the total number of
        assignment statements processed
        (i.e. $j$ is counter value for $V$.)
```

Then in all the successors — for each variable V, replace the corresponding argument of the ϕ-function(s), if any by a version V_i where i is at the top of the stack for V. After all the children of X in the dominator tree have been processed recursively by renaming, pop the stack for V, where V is the target of original assignment statement A and do it for all V and for all such A in the current node X.

These two algorithms generate minimal SSA form. For proof the reader is referred to [C$^+$91].

2.1.4 Example

In the given program flow graph of figure 1.5(a) $G = \langle N, E, n_0 \rangle$, $N = \{ b_1, b_2, b_3, b_4, b_5 \}$ and $n_0 = b_1$. The iterated dominance frontier of each node is as under :

$DF^+(b_1) = \{\}$
$DF^+(b_2) = \{b_2\}$
$DF^+(b_3) = \{b_2, b_5\}$
$DF^+(b_4) = \{b_2, b_5\}$
$DF^+(b_5) = \{b_2\}$

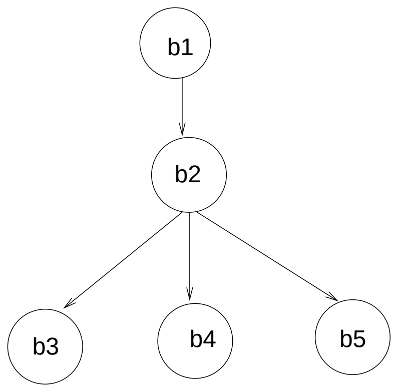

Figure 2.1: Dominator tree of figure 1.5(a)

The dominator tree with the original program flow graph are shown in figure 2.1. The SSA form is in figure 1.5(b). The *ϕ-Insertion* algorithm inserts ϕ functions at the entry of the nodes b_2 and b_5, as nodes b_3 and b_4 contain definitions of a and nodes b_2, b_5 are elements of iterated dominated frontiers of node b_2 and b_3. ϕ-function is not inserted anywhere due to the definitions of a and b in node b_1 as $DF^+(1)$ is empty. Then *Renaming* algorithm starts from node b_1 in the dominator tree constructed and makes a preorder pass. Due to preorder pass, the target of ϕ-function in node b_2 becomes a_1. As node b_5 is visited after node b_3, the target of ϕ-function is in node b_5 is changed to a_3 and one argument of ϕ-function in node b_2 is made a_3.

2.1.5 Bit Vector Based *Versus* SSA Based Analysis

Here we are presenting a comparative study of Bit Vector analysis and SSA based analysis.

- *Number of Def-use Chains* : The number of def-use chains in SSA form of a program is considerably less than that in the conventional intermediate forms. SSA attempts to minimize the number of def-use chains.

- *Computational Complexity* : The number of iterations required by bit-vector based algorithms to achieve MFP depends on the back edges. But in SSA, the effect of loop and back edges are removed due to the ϕ-functions and the single assignment property. So any SSA optimization guarantees linear time complexity in terms of the number of nodes in the program flow graph. This can be proved by the fact that for a loop, the *ϕ-insertion* algorithm inserts a ϕ at the beginning of the loop entry node, if a node in the loop contains an assignment to a variable. One of the argument of this ϕ-function is the version of the changed variable, which is visible at the end of the exit node of the loop. Thus in one pass of the program flow graph in SSA form, any change due to the back edge can be taken care of while computing the data flow property in question. In short, SSA form makes the program *flow-insensitive* with respect to the back edges and thus guarantees linear time complexity.

- *Processing Cost and Memory Requirement* : Bit vectors are generally computed for entry and exit points of all the basicblocks. Therefore the processing cost and memory requirement for bitvector frameworks depend on the number of basicblocks. But SSA is a sparse graph representation and only deals with objects where they occur or get changed. So cost of processing SSA form is less than the cost of processing bit vectors in most cases. From the beginning of analysis, bit vectors must be initialized with proper value. But in SSA, the program entity comes into existence only when it actually occurs in the program, thus reducing the memory requirement.

- *Name-space problem* : In SSA due to renaming of variables, a large number of names are used and the symbol table must handle all of them effectively. Here care must be taken to avoid any conflict that arises due to variable names. This problem does not occur in bit vector analysis.

- *Size of intermediate representation* : The size of intermediate representation in SSA increases due to introduction of new variables and ϕ-functions.

2.2 SSAPRE : Partial Redundancy Elimination Using SSA

SSAPRE is a formulation of lazy code motion [KRS92] in SSA and therefore uses some of the terminology of LCM. It uses *DownSafety* to denote the anticipatability condition, which is same as *D-Safe* or *Down-Safe* concept of LCM. Conceptually, SSAPRE hoists the partial redundant expressions to the latest program point as done in LCM to satisfy life-time optimality. SSAPRE algorithm uses six separate stages in all to eliminate partial redundancies. They are Φ-Insertion, Renaming, DownSafety, WillBeAvail, Finalize and CodeMotion. The first two stages are applied once on the program and last four stages are applied expression by expression.

2.2.1 Algorithms of SSAPRE

As pointed out by [DRZ92], detecting common expressions in a program in SSA form is difficult as they are hidden due to the different versions of variables involved in the

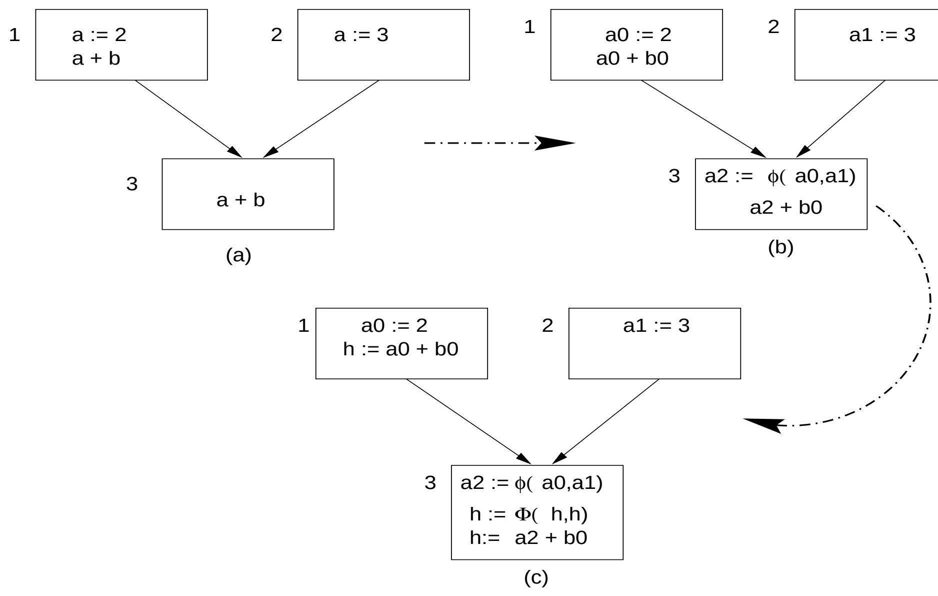

Figure 2.2: Use of Φ-functions and h; (a) Original program fragment. (b) SSA form of it and (c) the program fragment after Φ-Insertion of SSAPRE.

expression. Chow *et al* [C$^+$97] added a hypothetical temporary h^e for an expression e which will act as the its expression variable.

In the first stage Φ-*Insertion*, Φ-function for an expression e is introduced in the iterated dominance frontiers of the nodes containing an original occurrence of the e. SSAPRE also inserts Φ-functions at the points containing ϕ for any variable[3] that is an operand of e, only if e is partially anticipatable at that point. This technique is almost same as that of the ϕ-insertion for variables. The algorithm for ϕ-insertion can be found in [C$^+$91]. Note that a Φ is inserted for an expression whereas a ϕ is inserted for a program variable.

2.2.1.1 Example

The program flow graph of figure 2.2 contains an expression $a + b$ in node b_1 and b_3 and it is partial redundant in node b_3. (b) part of the figure 2.2 presents the standard SSA form. In part (c), the hypothetical variable h is inserted as the target of the expression $a + b$. A Φ-function is inserted after the ϕ-function of node b_1. The dotted arrow denotes the flow between different steps. SSAPRE inserts Φ-functions for an expression and then uses the algorithms to remove any partial redundancy involving that expression.

[3]A variable in SSA form of a program means a version of the original variable

The second stage *Renaming*, assigns versions to h^e for an expression e in a manner almost same as renaming technique for normal variables. A new version to h^e is assigned if there is an assignment to an operand of e after the previous occurrence of e and before the current occurrence. A Φ-function is always given a new version. If along a predecessor of a node x, there is no occurrence (neither original nor Φ-occurrence) after an assignment to an operand, a special version $\perp$ is assigned to the corresponding operand of the Φ-function in x if any. During renaming stage, SSAPRE sets two flags for their use in the later stages. These two binary flags are *has_real_use* and *down_safe*. Initially the flags *down_safe* is initialized to true and *has_real_use* is initialized to false. *down_safe* is for a Φ-function of an expression. A false value of *down_safe* means that the expression is not anticipatable at the entry of the node containing the Φ-function. Therefore the *down_safe* flag give the information same as the anticipatability at the entry of the nodes.

The *has_real_use* flag is for each Φ-operand of a Φ-function of an expression. Iff value of *has_real_use* is true for a Φ-operand of an expression in node x, then along the predecessor corresponding to the operand, the expression has an original occurrence and it is reaching at the entry of x. SSAPRE *incorrectly* sets these flags and thus computes erroneous DownSafety. We have discussed the cases where SSAPRE fails in section 2.2.2.

The third stage *DownSafety* computes anticipatability at the points having Φ-functions for the expression e. Computing anticipatability is a backward problem. Therefore this stage takes a backward propagation along the *def-use* chains of operands of the Φ's for which *down_safe* is false. The algorithm starts with a Φ-occurrence in the program and if it is not down_safe then recursively resets its operands' down_safe flag. The down_safe flag of an operand is reset only if it has no real use (*i.e. has_real_use* is false along that operand) and it is defined by Φ-function.

The fourth stage *WillBeAvail*, computes four predicates *can_be_avail*, *later*, *insert* and *will_be_avail*. *WillBeAvail* takes two forward passes on the program. In the first pass it computes for each Φ the can_be_avail flag. A Φ-occurrence is can_be_avail at a point p, if and only if the expression is anticipatable at p (*i.e. down_safe*) is or it has no $\perp$ operand or both. In the second pass only those Φ expressions are taken into account for which can_be_avail is true. Only in such cases insertion can make the expression totally redundant and thus that can be eliminated. Due to *Lazy Code Motion*, the code is inserted at the latest point in the program. This is done by computing *later* predicate. A Φ has *later* true if for the expression *can_be_avail* is true and each of its operands is non-$\perp$ or its *has_real_use* is false. An insertion point satisfies will_be_avail and not *later*.

The fifth stage *Finalize* inserts computations in the points at which *will_be_avail* is true and correspond to Φ-operands at which the expression is not available. It removes all the extraneous Φ's. It also finds whether the occurrence of h version is a def or use, since this is used to insert actual temporaries. *Finalize* performs a preorder traversal of the dominator tree and visits a defining occurrence h that has to be saved in a temporary t_x, where x is the version of the actual expression temporary. Note that finalize does not places t_x in the program. It only finds which occurrences of h must be saved and which occurrence must be reloaded from an already computed version of h. For this at each visit to a node, two predicates *save* and *reload* are computed. Preorder traversal is needed as it has to decide which occurrence has to be saved, before visiting the use of this definition. If for a Φ occurrence encountered, will_be_avail is true, that means the corresponding h_x is visited for the first time, this definition of h_x is kept track of, so that it can be found out whether this is used somewhere later in the program or not. For an original occurrence, if we are visiting h_x for the first time or an occurrence that is not dominated by a previous one, then these are new definitions of h. But if none of these is true for the real occurrence, then this is a use of h_x and so save flag is set to true for the previous occurrence (which makes current occurrence totally redundant here). At the same time, reload flag is set to true for the current occurrence, since the computation has to be removed and the previous result is to be used at this point. Each occurrence of h_x (which occurs in a predecessor block of b) as an operand of a Φ in block b is processed as follows. If will_be_avail flag is true for this Φ (implies that insertions shall make this occurrence redundant), then for each operand Insert flag is checked. If Insert is true, a new computation is inserted at the exit of the corresponding predecessor of b.

In all the previous stages, the SSA form was not a valid one due to the $\perp$ operand and due to the hypothetical temporary h and its Φ functions. The program remains in valid SSA form after CodeMotion stage. CodeMotion traverses the graph for h and assigns a new temporary version t_i to those occurrences of the expressions, for which *save* is true and replaces a computation by a load of the temporary, if *reload* flag is true. At the Φ of h it generates a ϕ for t. If an original occurrence is encountered by CodeMotion while visiting the SSA graph and if *save* is true, then this is a new computation. So a new version of t is generated to hold the value of the computation. For reload is true, the corresponding version replaces the computation. At an inserted occurrence, the result is saved into a new version of t and at a Φ occurrence, a ϕ for t is placed. This finishes the CodeMotion step.

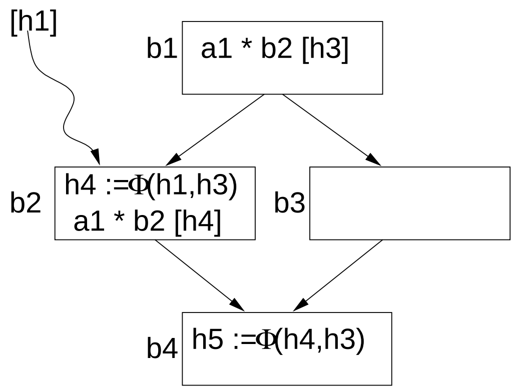

Figure 2.3: Illustration of a case in which SSAPRE computes *has_real_use* incorrectly

2.2.2 Correctness Problem with SSAPRE

We have detected that SSAPRE computes two flags *has_real_use* and *down_safe* incorrectly. The figures 2.3 and 2.4 are the examples respectively. In the figures we have ignored the ϕ-functions for the variable(s), if any.

2.2.2.1 Problem and Solutions for correctly Computing *has_real_use*

In figure 2.3 the nodes b_2 and b_4 contain a Φ-function each for expression $a * b$. b_4 is the exit node. b_2 also contains a an original occurrence $a_1 * b_2$. As is clear from the figure, *has_real_use* of the first operand h_4 of the Φ-function in node b_4 is true. This is because there is the real occurrence $a_1 * b_2$ with same version h_4 in node b_2 along the path $[b_2, b_4]$. But SSAPRE does not set the *has_real_use* of h_4 to true during renaming stage. This is because while the first operand of h_5 Φ-function is given version from b_2, the top of the renaming stack of $a * b$, contains h_4 as the Φ-function and checking for if the top of stack is an original occurrence fails. It is noteworthy here that SSAPRE [C$^+$97] follows Cytron et al's algorithm and therefore pushes only new versions onto the stack.

This can be solved by the simple method : push all the occurrences of the expression encountered during renaming to the stack. This will ensure that the checking for an original occurrence on top of stack while assigning a version to the Φ-operand will give correct result and therefore the *has_real_use* flag can be computed correctly.

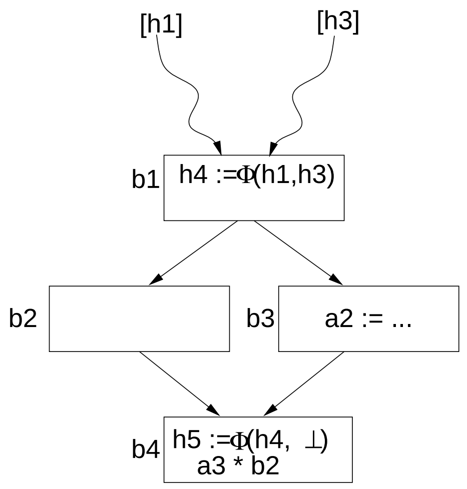

Figure 2.4: Illustration of a case in which SSAPRE computes *down_safe* incorrectly

2.2.2.2 Problem and Solutions for correctly Computing *down_safe*

One of the reason why SSAPRE computes *down_safe* flag incorrectly is the incorrect value of *has_real_use* flag computed during renaming. The other problem is illustrated by figure 2.4. The node b_4 contains a Φ-function h_5 at which the expression $a * b$ is anticipatable due to the original expression $a_1 * b_2$ at the end of b_4. But the expression is *not* anticipatable at the Φ-function h_4 in node b_1. This is due to the assignment to the operand a in node b_3. But SSAPRE does not reset the value of *down_safe* flag for h_4 Φ-function. This is because SSAPRE resets the *down_safe* flag of a Φ-function if

- while giving a new version to an original occurrence, the top of the expression renaming stack contains the Φ-function or
- while exit point of the program is reached, the top of the renaming stack contains the Φ-function.

But in the case given by the figure 2.4, neither of the above conditions satisfy and therefore SSAPRE does not reset the *down_safe* flag for the Φ-function in b_1. In the figure, h_4 is not used as a Φ-operand in any other Φ-function and therefore in the *DownSafety* stage also its *down_safe* flag will remain true.

This can be solved as follows : whenever a Φ-operand is assigned a version $\perp$, check the top of expression renaming stack. If the top of the expression renaming stack contains a Φ-function $h_i \leftarrow (\ldots)$, then reset the *down_safe* flag of h_i to false. This will take care of the above case as well as all the cases in which anticipatability at a point becomes false.

2.2.3 Analysis of SSAPRE

SSAPRE suffers from the correctness issue of computing anticipatability. It also pre-splits critical edges before applying Φ-insertion. Therefore the size of the program almost doubles. For this reason, SSAPRE inserts computations in Finalize stage along edges, that are critical *i.e.* in the inserted nodes in the edges. It therefore inserts more number of computations than required. If along all the outgoing edges of a node insertion has to be made, then instead of inserting along all edges, exit of the node can be a better placement point. Thus the size of the optimized code (which is important in embedded systems and in many applications) is suboptimal. Also the algorithm hoists and then later sinks the computations to attain life-time optimality of LCM. This is an expensive operation. SSAPRE algorithm is also pretty complex.

2.3 Strength Reduction of Large Expressions

A large expression involves more than one operators as defined in [DD95]. The SRLE algorithm developed by Dhaneshwar and Dhamdhere unifies code hoisting with the strength reduction. Strength reductions of subexpressions containing one high strength operator each of the large expression may not give the maximal optimization. This can be illustrated by the following example. The program of figure 1.2 computes $i * c1 + j * c2$. The three-address code is given in the program flow graph. In the first optimization (strength reduction using conventional technique) the number of add operations in the loop is two (see figure 1.3. But if large expressions are optimized as a whole, then the number of add operators becomes one, which is a significant reduction in cost due to the possible repetitive execution of the addition operations in the loop. The srle-optimized program is shown in figure 1.4.

The difference between the X-properties and the sr-properties is that in the former one any number of update computations for a single variable along a path can be ignored. But in the later case only at most one update computation is ignored. This allows the SRLE algorithm to overcome the inefficiency faced by CHSA [JD82, Dha89]. Also the algorithm uses EPA synthetic node insertion for maximal redundancy elimination. The placement criteria of SRLE [DD95] are reproduced here again.

2.3.1 SRLE Placement Criteria

The following placement criteria for normal PRE, given by DPH–1 and DPH–2 and SRLE given by DPS–1 to DPS–4 are reproduced from [DD95].

[DPH–1] A computation of e is placed at the exit of node b_l iff

1. e is not available at the exit of b_l,
2. e is not eliminatable in b_l, and
3. All paths starting at the exit of b_l have a prefix $[b_l \dots b_k]$ such that b_k contains an eliminatable occurrence of e, and for all nodes b_j on the path $(b_l \dots b_k)$, e is eliminatable in b_j and b_j is empty with respect to e.

[DPH–2] A computation of e is placed in a synthetic node b_{l-m} inserted on the edge (b_l, b_m) iff

1. e cannot be placed in b_l due to the violation of condition DPH–1(3),
2. e is neither available nor eliminatable at the exit of node b_l, and
3. All paths starting at the entry of node b_m have a prefix $[b_m \dots b_k]$ such that b_k contains an eliminatable occurrence of e, and for all nodes b_j on the path $[b_m \dots b_k)$, e is eliminatable from b_j and b_j is empty with respect to e.

The criteria for placement of recomputations are given by DPS-1 and DPS-2. The criteria placement of update computations are given by DPS-3 and DPS-4.

[DPS–1] A recomputation of e is placed at the exit of node b_l iff

1. e is not sr-available at the exit of b_l,
2. e is not sr-eliminatable in b_l, and
3. All paths starting at the exit of b_l have a prefix $[b_l \dots b_k]$ such that b_k contains an sr-eliminatable occurrence of e, and for all nodes b_j on the path $(b_l \dots b_k)$, e is sr-eliminatable in b_j and b_j is X-empty with respect to e.

[DPS–2] A recomputation of e is placed in a synthetic node b_{l-m} inserted on the edge (b_l, b_m) iff

1. e cannot be placed in b_l due to the violation of condition DPS–1(3),
2. e is not sr-available at the exit of node b_l, and
3. All paths starting at the entry of node b_m have a prefix $[b_m \dots b_k]$ such that b_k contains an sr-eliminatable occurrence of e, and for all nodes b_j on the path $[b_m \dots b_k)$, e is sr-eliminatable from b_j and b_j is X-empty with respect to e.

[DPS–3] An update computation of e is placed at the exit of node b_j containing an update definition of e, which is not followed by a recomputation of e iff b_j lies on a path $[b_l, \ldots, b_k)$ in the optimized program such that

1. A locally X-available occurrence of e (original computation or inserted computation) exists in l.
2. b_k contains an sr-eliminatable occurrence of e.
3. All nodes on path $(b_k, \ldots, b_k)$ are X-empty with respect to e.

[DPS–4] An update computation of expression e is placed at the entry of node b_k iff b_k contains an update definition of an operand of e preceding an sr-eliminatable occurrence of e.

2.4 Strength Reduction Using SSAPRE

Chow *et al*'s [K$^+$98] strength reduction unified to the SSAPRE actually uses the NCHSA of [Dha89] . In this technique they call the update definition as *injuring* definition and a update computation as an *injured* computation. recomputation/redefinitions are called killing computations/definitions. Only three stages of SSAPRE need to be changed to accommodate strength reduction. They are described briefly in next sections.

2.4.1 Φ-Insertion

In this new version of this algorithm all the injuring definitions of a variable are bypassed till a version is reached which is not defined by a update definition. This is the improvement to the part pertaining to the iterated dominance frontier node collection. For Φ insertions for the operand definition by a ϕ, again similarly injuring definitions for the same version is bypassed until a version is reached which is not defined by an injuring definition. If such a version is defined by a ϕ, then as usual it is entered in list for being considered for Φ insertion.

2.4.2 Renaming

Renaming has to be modified after Φ-insertion is modified to take care of injuring definition (or update definition). The most crucial modification in this stage is as under :

If q is the occurrence of the expression and any one version on top of the rename stack of variables does not match with that of the version at the top of the expression rename stack, then find out whether v_q is defined by an injuring definition. If so, then give the expression a new version else recursively find out whether the right hand version of definition of v_q is defined by an injuring definition. If q is an injured occurrence, then assign q the same h-version as that of the top of the expression rename stack. If q is not an injured occurrence, follow the same procedure of standard SSAPRE *Renaming* stage; *i.e.* if q is a real occurrence, assign q a new version; if q is a Φ operand, assign the special version $\perp$ to that Φ operand to denote that no evaluation of expression E reaches that program point.

2.4.3 CodeMotion

This is final stage of the SSAPRE and strength reduction. The algorithm accumulates the update value until the next injury is found and the insert points for update computation is identified by a flag need_repair. The flag is true at the points containing closest injured definitions to at least one use of t. Then the insertion is done.

2.5 Analysis of Strength Reduction Frameworks

The SSA-based strength reduction framework of [K+98] is an algorithm with linear complexity. The drawbacks of strength reduction using SSAPRE are

- Multiple update computations are inserted on a path; thus it may increase the execution time on that path. This is the same inefficiency, CHSA suffers from.
- It does not take care of mutual induction variables.
- It gives suboptimal result for strength reduction of large expressions.
- Since this scheme is based on the SSAPRE of [C+97] and since SSAPRE computes some of the properties incorrectly (see section 2.2.2), this scheme for strength reduction will also give incorrect results.

An example of mutual induction variables is given in the following program fragment.

```
for (k = 0; k < 10; k++) {
    i = j + 2;
    ...
    j = i + 5;
}
```

In this i and j are mutual induction variables.

Chapter 3

Overview

We have developed algorithms for SSA based partial redundancy elimination [KD]. We call this framework as E-path_PRE as it uses e-paths. Our formulation of strength reduction of large expressions in SSA is called as E-path_SRLE. In ensuing sections, we are presenting an overview of E-path_PRE and E-path_SRLE.

3.1 Overview of E-path_PRE

The input to our E-path_PRE [KD] algorithm is the SSA form of the program and output is the SSA form of the PRE-optimized program. It solves PRE using eliminatability paths to detect partially redundant computations. E-paths are computed with respect to an expression. For the purpose of efficient handling of e-paths they are collectively represented in the form of a graph called eliminatability graph (e-graph). Each maximal path in this graph is an e-path with respect to the corresponding expression. By the definition of e-paths all the empty nodes those are part of the e-graph have to be safe at their exits. This needs computation of availability and anticipatability at the exit of the nodes. After getting the e-graph insertion points are computed and then insertion of computations is performed. The last step eliminates the now-redundant computations. Since we are using SSA form for PRE, the optimized program has to be standard SSA form.

Since detection of lexically identical expressions in the SSA form of the program is difficult, we compute PRESSA form of the program prior to any computation, in which each expression e is associated with a hypothetical variable h^e. In the PRESSA form, at the join nodes where more than one values of h^e are possibly reaching, a Φ-function for e is inserted. The notation Φ is used to avoid confusion with ϕ-functions — Φ-function is for expression variable h^e and ϕ-function is for original program variables.

Our approach solves the problem of partial redundancy elimination (PRE) using six stages [KD] broadly (or eight separate stages) as under.

1. PRESSA form
   (a) Φ-Insertion
   (b) Renaming
2. Computation of Safety
   (a) Computation of Availability
   (b) Computation of Anticipatability
3. Computation of Eliminatability Paths
4. Computation of Insertion Points
5. Insertion
6. Elimination of Redundant Computations

The first four stages namely Φ-insertion, renaming, computation of availability and anticipatability are applied only once to the program and last four stages are applied expression by expression. We present the flow of the stage application in figure 3.1. In the following sections we are giving brief overviews of the stages.

3.1.1 Computation of PRESSA form

At the entry of a node x, a Φ-function for h^e is inserted iff x is a loop entry node or x is an element of the iterated dominance frontier of another node y containing an original occurrence of e. Also a Φ-function foes h^e is inserted at the entry of x contains a ϕ-function for an operand of e and e is partially anticipatable at the entry of x.

Renaming stage assigns versions to the hypothetical variable h^e of expression e. A preorder traversal of the dominator tree is performed and new versions are given to h^e. A Φ-function is always given a new version. A real occurrence is given a new version if after the last occurrence there is an assignment to an operand. A Φ-operand is assigned a special version $\perp$ if along the corresponding predecessor, the expression is not available. During renaming, whether a node is an empty node with respect to an expression, or the expression is locally anticipatable or contains an available at exit occurrence or both is also collected for each node for each expression. Boundary values for availability and anticipatability at

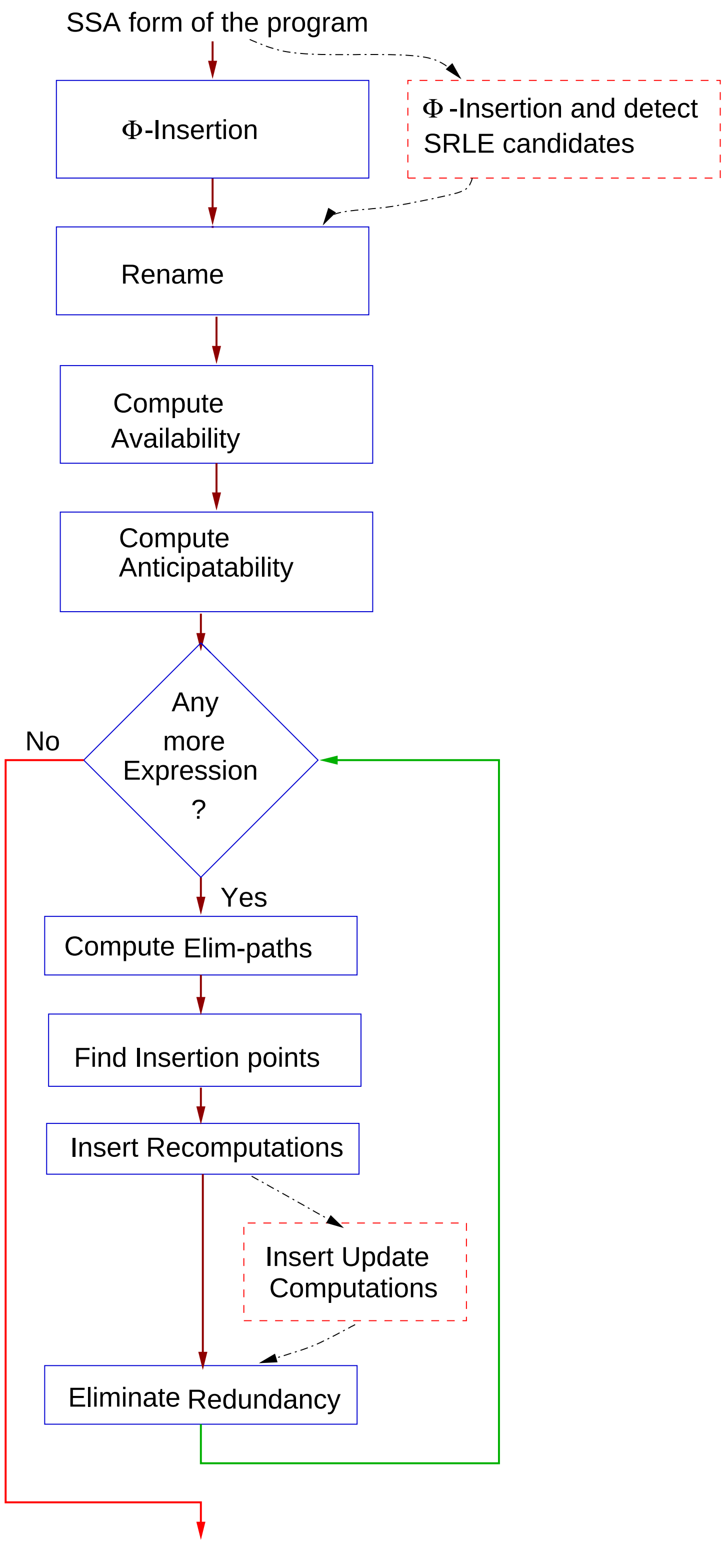

Figure 3.1: The stages of our approach for solving PRE. As shown, the last four stages are performed on an expression by expression basis. The dotted basicblocks are illustrating the stages of E-path SRLE expressions.

the nodes containing Φ-functions are computed during renaming. For propagation of these values to other nodes, a property called *has_real_use* is also computed.

3.1.2 Computation of Availability

Initially the boundary values of availability that are false are propagated along the def-use chains to other nodes containing Φ-functions for the same expression if *has_real_use* is false along that path. Then computation of availability is performed through a preorder traversal on the dominator tree of the PFG. Availability at the entry and at the exit of the node are computed. Since availability is a forward problem, property at entry is propagated to the exit of the node using the local information collected during Renaming stage. If a node contains no Φ-function, then the availability at its entry is same as availability at the exit of its immediate dominator. If a node contains a Φ-function, then the availability at the entry is same as that of the availability due to the Φ-function.

3.1.3 Computation of Anticipatability

Initially the boundary values of anticipatability that are false are propagated to other node entries containing Φ-functions for the same expression if *has_real_use* is false along that path. This is same as the DownSafety pass of SSAPRE. Next a postorder traversal of the dominator tree is made to compute anticipatability at the entry as well as the exit of all the nodes. Since anticipatability is a backward problem, the value of anticipatability at the exit is propagated to the entry depending on the contents of the node (during renaming we already have computed what type of the node is). While computing anticipatability at the exit of the node, if anticipatability at the entry of a successor is false, then the value is made false. Otherwise if a successor has not been visited and it is not a loop entry node, then recursively anticipatability at the entry of the successor of the node is computed through a recursive call to this algorithm and then the normal postorder traversal is resumed. If a successor is a loop entry node then the value of anticipatability at the entry of the successor is the logical AND with the current value.

3.1.4 Computation of eliminatable paths

To compute e-paths, the algorithm makes a preorder traversal on the dominator tree of the PFG to build the eliminatable paths. At the end of the pass the output is a graph with similar structure of PFG but possibly having multiple entries and exits. While making the traversal, the addition of nodes is done in a demand-driven manner. A node is added to the graph being constructed iff a successor or a predecessor of the node along with itself satisfy the constraints as defined by e-path. While a node x gets added to the graph, it gets splitted to x_l^{in} and x_l^{out} if it has got both an anticipatable occurrence and an available at exit occurrence of the expression. But these two nodes are added separately to the graph, only if when they are needed. During the computation of the egraph, some nodes those cannot be part of any e-path may have been added. Therefore such nodes (called useless nodes) are removed from the graph after computation of all the e-paths.

3.1.5 Computation of Insertion Points and Insertion

Insertion point is an edge (x,y), iff y has a corresponding node in e-graph and this edge is not in the e-graph while availability of the expression e in question at the exit of x is false.

A preorder traversal of the dominator tree is performed for insertion. A computation is inserted at the exit of a node iff all the outgoing edges of the node have been marked as insertion points otherwise the edges those are marked are splitted and a synthetic node is inserted along the edge. During the traversal versions of the expression operands are collected and at the insertion point, proper version of the expression is inserted. To reduce the size of SSA graph, extraneous Φ's are also detected and the immediately following real occurrences that are uses of these Φ-functions are given new versions. In addition to this, all other uses of this Φ-function are replaced by the version that dominates this use.

3.1.6 Elimination of Redundant Computations

In the Elimination of Redundancy phase, extraneous Φ's are removed. Then the SSA graph of h is traversed and all definitions of h are given new versions of temporary t and all its uses are removed by the reload of this version of the temporary. The corresponding Φ-functions are replaced by ϕ-functions. Then at the end, all remaining h versions and Φ functions will get removed from the final graph.

3.2 Overview of E-path_SRLE

The input is the program flow graph in standard SSA form. The output is the SSA form of the SRLE-optimized program. An SRLE-optimized program contains no eliminatable partially redundant computations and no strength-reducible expressions, including large expressions. Initially SRLE candidate expressions are detected and then SRLESSA form is computed. For computing SRLESSA form $\Phi_{1,2}$-functions are inserted. Renaming is performed as per in the E-path_PRE except the s in the ordered pair (h,s) (see section 1.3.2, is renamed by bypassing at most one update definition for an operand. Renaming is followed computation of srle-availability and normal availability. Computation of srle and normal anticipatability follows availability stage. The e-path computation algorithm is applied for expression e after being preceded by checking whether e is a proper SRLE candidate. If not, then normal PRE stages will be applied just after replacement of Φ_2-functions for e if any there by Φ_1-functions. The insertion of update computation is the strength reduction stage.

E-path_SRLE has got one more stage apart from the eight stages of E-path_PRE. The new stage is insertion of update computations that precede elimination of redundancy. Figure 3.1 depicts the ordering of stages. The dotted rectangles are stages of SRLE and the dotted arrows are the flow of control for SRLE. The last five stages are applied to the program expression by expression and the first four stages are applied once to the whole program. Also for expressions that are not proper candidates of SRLE four stages of e-path_PRE are applied thereby reducing the cost of optimization.

Chapter 4

Algorithms of E-path_PRE

In this chapter the algorithms of E-path_PRE [KD] have been discussed, analyzed and illustrations of different stages have been presented. Correctness proofs of algorithms and some important lemmas have also been included in the discussion. The implementation of E-path_PRE has also been discussed briefly.

4.1 Φ-Insertion

A Φ-function for each expression e is inserted in each node which is in the iterated dominance frontier of a node containing an occurrence of e [C$^+$91, JPP94, SG88] and in each node that contains a ϕ-function for an operand of e. In addition, we also insert a Φ-function in the entry node of a loop to simplify the computation of anticipatability.

4.1.1 Example

The program in figure 4.1 is used for explaining and analyzing all the stages of E-path_PRE. The dominator tree is shown in figure 4.2. The dominance frontier of each node is shown in the dominator tree adjacent to each node. This tree is used almost in all stages of our algorithm except in the last stage. The standard SSA form of the program of figure 4.1 is shown in figure 4.3. This form of the program is the input to our algorithm. The program after Φ-insertion is shown in figure 4.4.

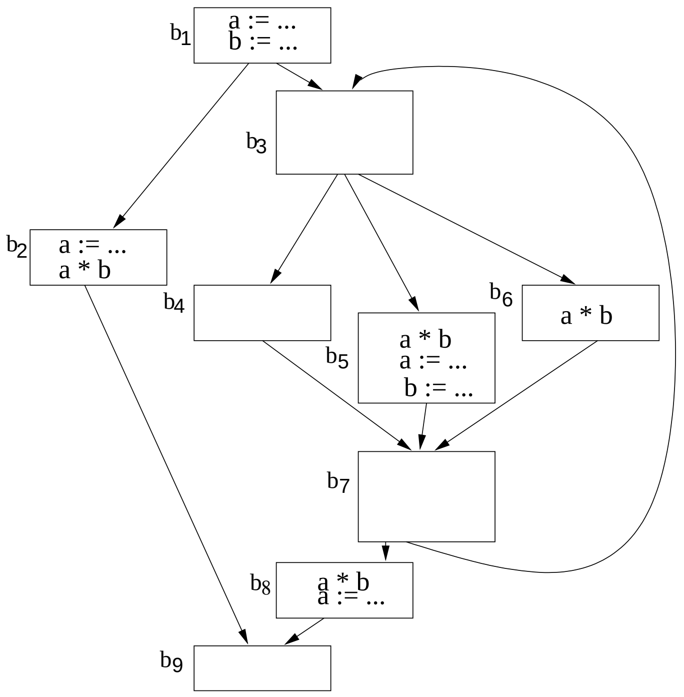

Figure 4.1: Standard PFG used for explanation

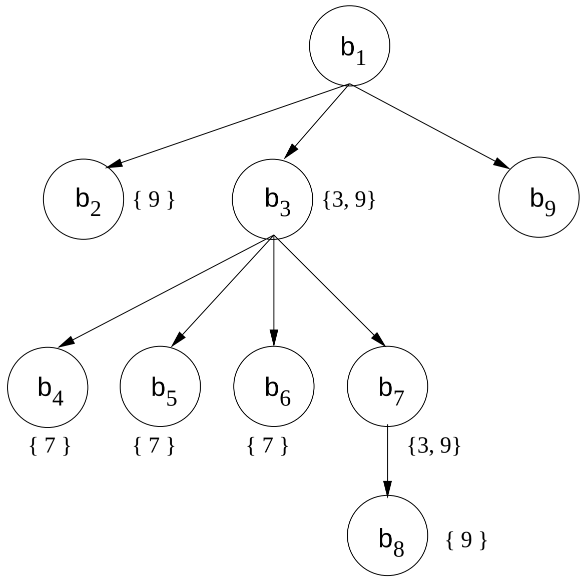

Figure 4.2: Dominator tree of the PFG used for explanation. Near each node the set of dominance frontier are shown.

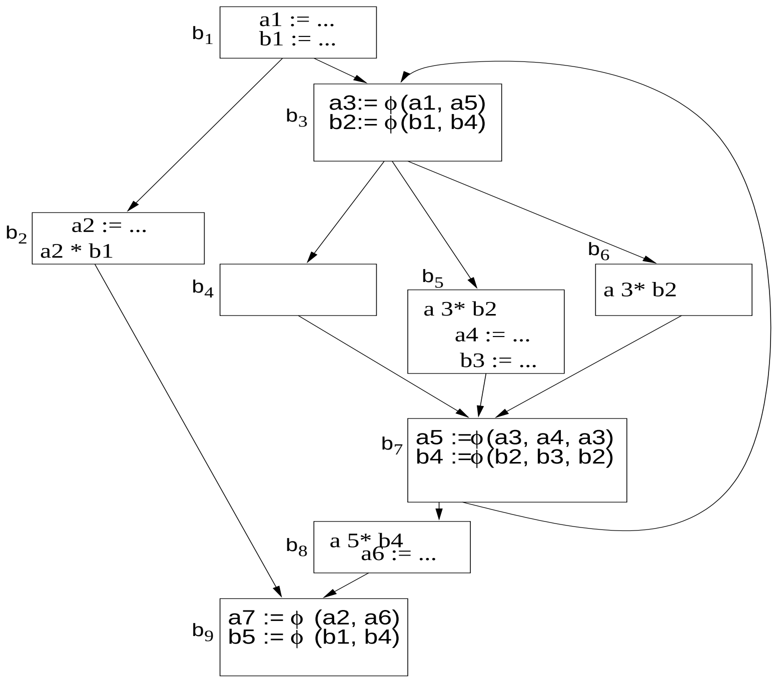

Figure 4.3: SSA form of the program used for explanation

4.2 Renaming

This step performs renaming of the expression variables, *i.e.* variables of the kind h^{a*b}, by assigning them version numbers. As mentioned earlier, every occurrence of an expression $a * b$ is considered to be an assignment $h^{a*b} \leftarrow a * b$. If renaming were to be done as in the SSA form, each occurrence of $a * b$ would create a new version of h^{a*b}. This is not always necessary. Hence we assign a new version number at an occurrence of $a * b$ only if, along some path reaching it, an assignment to either a or b has occurred after the last occurrence of $a * b$. We also assign a new version to every Φ-function. We call the program form after Φ-insertion and renaming as PRESSA [KD] form to distinguish it from the SSA form. This stage also computes four flags to be used in later stages - *has_real_use*, *available*, *down_safe* and *node_type*. They are used for computations of availability, anticipatability and e-graph. The first three flags are binary (true or false) flags. *node_type* is to indicate local properties of a node. This flag takes the following values:

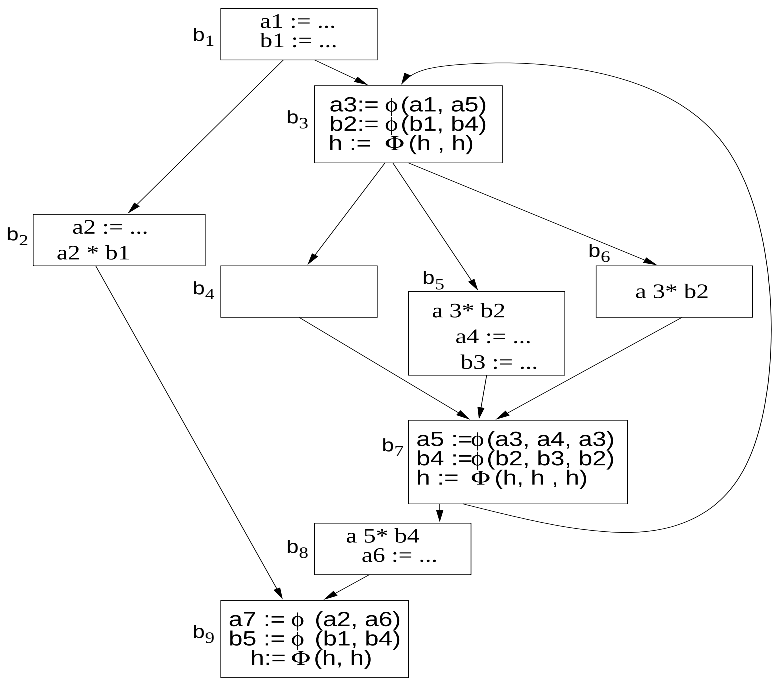

Figure 4.4: The program after Φ-Insertion.

empty	node x is empty with respect to expression e
antloc	e is locally anticipatable but not locally available
avail	e is locally available but not locally anticipatable
both	if x is both *antloc* and *avail*
others	otherwise.

Renaming is performed in a preorder traversal of the dominator tree. We maintain renaming stacks for all operands of an expression, as also its result name (that is the hypothetical variable h^{a*b}). The renaming technique is based on that of Cytron et al's technique for renaming of program variables. An entry in the renaming stack for a result name indicates the version numbers assigned to its operands as well as the version number of the result name. The renaming stack of h^{a*b} is called as expression renaming stack. In the SSA form of a program there are three types of occurrences of an expression - original occurrence (occurrence that occurs in the original source program), occurrence as a Φ-function and occurrence as a Φ-operand. The first two can be defining occurrences of the expression. Versions are assigned as per the following :

- A Φ-function is always given a new version.
- A real occurrence at a point p is given a new version iff along some path from entry to p, there is an assignment to an operand after the previous occurrence of the expression; *i.e.* there is a mismatch between the operand versions at the top of the expression renaming stack and the versions at the top of the corresponding operand-renaming stacks. If there is no such mismatch, then the most recent version of h^e, that is on top of the expression renaming stack is assigned to this original occurrence.
- A Φ-operand is given the same version as that at the top of the expression renaming stack, if there is no mismatch between the operand versions at the top of expression renaming stack and those at the top of operand-renaming stacks. If there is a mismatch, this ensures that along that path the expression is not available and therefore is assigned a special version $\perp$.

While assigning a new version to a defining occurrence of the expression, the operand versions as well as the new version of h^e are pushed onto the expression renaming stack. If an original occurrence is not given a new version but is assigned the most recent version, then also it is pushed onto the expression renaming stack. This is where our algorithm of renaming is different from the renaming algorithm of Cytron et al and that of SSAPRE. Both of them push the version if it is a new version. We push all the versions given to any defining occurrence to the expression renaming stack, thereby keeping track of all the occurrences encountered along a path. This allows the renaming algorithm to correctly compute the *down_safe* and *has_real_use* flag, which are *incorrectly* computed by SSAPRE as mentioned in section 2.2.2. After all children of a node in the dominator tree have been visited, the entries that have been added to the renaming stacks during processing of the node are popped off.

4.2.1 Computation of Flags

Initially the flags *down_safe* and *available* are initialized to true. Each of these two flags are for each Φ-function of an expression. A false value of *down_safe* (*available*) means that the expression is not anticipatable (available) at the entry of the node containing the Φ-function. Therefore *down_safe* and *available* flags give the information same as the anticipatability and availability at the entry of the nodes. The flag *has_real_use* is initialized to false. This flag is for each Φ-operand of a Φ-function of an expression. Iff value of *has_real_use* is true for a Φ-operand of an expression, then along the predecessor corresponding to the operand, the expression has a original occurrence and this Φ-operand has the same version as that of the original occurrence. *node_type* flag is initialized to *empty*.

We use the following method to determine whether a Φ-operand is a real occurrence[1] — While giving a version h_j to a Φ-operand of expression e, we check if the occurrence at the top of the expression renaming stack is an original occurrence with same version h_j. If this is the case, then we set $has\_real\_use_{h_j}$ = true.

Each operand of a Φ-function(henceforth called a Φ-operand) represents the value of e along some path β reaching the Φ-function. A Φ-operand is assigned the special version $\perp$ if β does not contain a Φ-function for e or an occurrence of e following last definition(s) of its operand(s). This indicates that e is not available along β. We set $available_{h_i}$ = false if a Φ-function $h_i \leftarrow \Phi(\ldots)$ has such an operand. We say an operand h_j corresponding to path β in a Φ-function $h_i \leftarrow \Phi(\ldots)$ 'is a real occurrence'[2], if an occurrence of e with version h_j not followed by definitions of operands of e or by a Φ-function exists along β. Such a Φ-operand indicates that value of e is available along β.

down_safe flag for a Φ-function for expression e is made false, iff the occurrence at the top of expression renaming stack is the Φ-occurrence and

- the exit of the program (that is exit point of exit node) has been reached or
- a new h^e version is being assigned to an original occurrence or
- $\perp$ version is being assigned to a Φ-operand (this may not be an operand in the same Φ-function).

SSAPRE does not take care of the last condition and thus erroneously concludes anticipatability as true at some points of the program whereas it is false at those points.

Our renaming algorithm computes *node_type* flag during renaming. By comparing versions of operands, it concludes the values of *node_type*. During computation, a temporary value *notantloc* is used for this flag to indicate that the expression is not locally anticipatable in the node. Figure 4.5 is the algorithm for computing the *node_type*.

4.2.2 Example

The example of figure 4.1 is used to illustrate the renaming scheme. The resulted PRESSA form of the example program is in figure 4.6. On the dominator tree (see figure 4.2), a

[1] that means whether *has_real_use* is true

[2] In this context only; in any other context, a real occurrence means the original occurrence in the program

```
Algorithm Compute_node_type(Node $x$)
{
    For each expression $e$ do node_type$_e$ $\leftarrow$ *empty*;
    At entry of node $x$ do {
        For each original occurrence of each expression $e$ do {
            If no original occurrence of $e$ has earlier been encountered in $x$ then {
                Compare versions of the operands of $e$ at the top of their
                variable-renaming stack with their versions at the entry.
                If there is no mismatch then *node_type*$_e$ $\leftarrow$ *antloc*;
                Else *node_type*$_e$ $\leftarrow$ *notantloc*;
            }
        }
    }
    At exit of node $x$ do {
        For each expression $e$ do {
            If $\exists$ a defining occurrence of $e$ in $x$ then {
                Compare operand-versions at the top of $e$'s renaming stack with
                those in variable renaming stacks;
                If there is no mismatch then
                    If (the last defining occurrence is an original occurrence) then
                        If (*node_type*$_e$ is antloc) then *node_type*$_e$ $\leftarrow$ *both*;
                        Else *node_type*$_e$ $\leftarrow$ *avail*;
                    Else *node_type*$_e$ $\leftarrow$ *empty*;
                Else
                    If (the last defining occurrence is an original occurrence) then
                        If (*node_type*$_e$ is notantloc) then *node_type*$_e$ $\leftarrow$ *others*;
                    Else *node_type*$_e$ $\leftarrow$ *others*;
            }
            Else {
                Compare current operand versions from their variable-renaming
                stacks with those at the entry of $x$;
                If there is no mismatch then *node_type*$_e$ $\leftarrow$ *empty*;
                Else *node_type*$_e$ $\leftarrow$ *others*;
            }
        }
    }
}
```

Figure 4.5: Computation of *node_type*. This algorithm is merged with renaming stage.

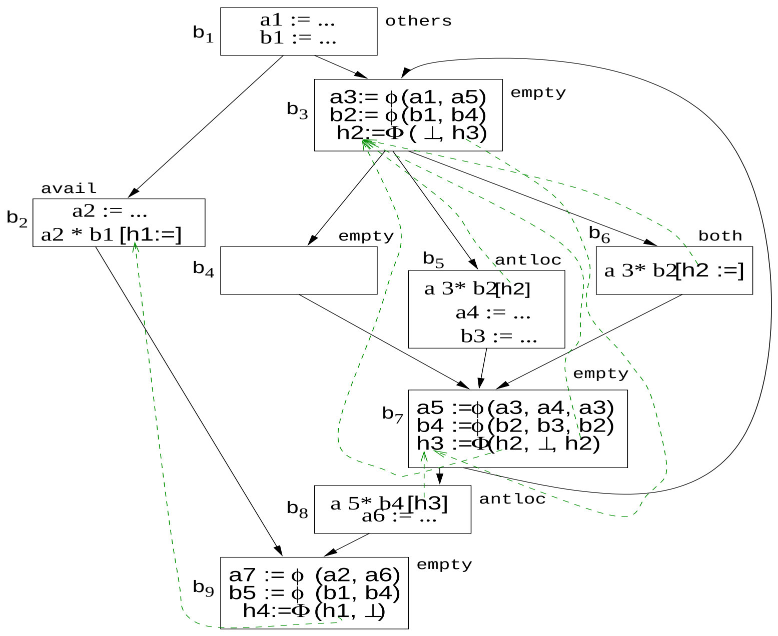

Figure 4.6: PRESSA form of the example program after Renaming. The values of *node_type* flag are shown adjacent to the corresponding nodes.

preorder traversal is performed. We have taken the assumption that starting version is always 1. In node b_2, the occurrence $a_2 * b_1$ is assigned a new version h_1 as the expression renaming stack is empty. The Φ-functions in b_3, b_7 and b_9 are given versions h_2, h_3 and h_4 respectively. In b_6, the original occurrence is given version h_2 as there is no mismatch between operand versions. Similarly occurrence in b_8 is given version h_3. The first Φ-operand in b_3 and the second Φ-operands in b_7 and b_9 are $\perp$ as along the corresponding first and second predecessors, the expression is not available (there is mismatch in operand versions).

The *node_type* flag values are shown in the figure 4.6 against the corresponding basicblocks. The *available* flag is false for Φ-functions in b_3, b_7 and b_9 as all of them contain $\perp$ as operand. *down_safe* flag is false for only b_8 as exit point is encountered. The *has_real_use* flag is true for third Φ-operand in node b_7 as along the predecessor b_6, there is an original occurrence reaching at the entry of b_7. Similarly *has_real_use* is true for the first Φ-operand in node b_9. The dashed lines in figure 4.6 are the def-use chains.

4.2.3 Analysis

Since our algorithm make s preorder traversal, the time complexity of the algorithm is linear with respect to the number of basicblocks in the program. The algorithm computes correct SSA form and computes the flag values correctly unlike SSAPRE.

4.3 Computation of Availability

Before we describe the algorithm of computation of availability, some theoretical results follow. These theoretical results simplify the computation of availability significantly.

Lemma 1 No path from the program entry node to node x contains an occurrence of e, if node x does not contain a Φ-function for expression e and no dominator of x contains either a Φ-function or an occurrence of the expression e.

Proof : Consider a dominator dom_x of x. Since dom_x does not contain a Φ-function, dom_x is not in the iterated dominance frontier of any node b containing an occurrence of e. Hence no path from the program entry node to dom_x contains an occurrence of e. Let some path from the program entry node to x contain an occurrence of e. This implies that the occurrence lies along a path from $Idom_x$, the immediate dominator of x, to x. If x has a single predecessor then x must contain an occurrence of e, which is a contradiction. If x contains > 1 predecessor, it must contain a Φ-function for e, which is also a contradiction. $\square$

Lemma 2 If a node x does not contain a Φ-function for an expression e, then availability at its entry is same as availability at the exit of its immediate dominator $Idom_x$.

Proof : The lemma is trivially true if x has a single predecessor p. Let x have > 1 predecessor. Let availability at entry of x not be same as availability at the exit of $Idom_x$. Hence there is at least one path $\alpha \equiv (Idom_x \ldots x)$ which either generates or kills the availability of e. If availability is killed along α, then α contains a definition of some operand v of e. This would lead to a ϕ-function for v in x which would give rise to a Φ-function for e in x, a contradiction. If availability is generated along α, then there is an occurrence of e along α, which will lead to a Φ-function in x for e, a contradiction. $\square$

The data structures used in the algorithms are the two binary flags *avail-at-entry* and *avail-at-exit*. These two indicate the availability at the entry and exit of a node with respect to an expression respectively.

Computation of availability needs a preorder traversal on the dominator tree of the PFG. Since availability is a forward problem, availability at entry is propagated to the exit of the node using local information (*i.e.* value of*node_type* flag) concerning the node. Following Lemma 2, if a node does not contain a Φ-function, then availability at its entry is same as availability at the exit of its immediate dominator. If a node contains a Φ-function $h_i \leftarrow \Phi(\ldots)$, then three possibilities exist concerning availability of the expression at its entry. Availability of e at entry to the node is false if $available_{h_i}$ = false. If a Φ-operand h_j is itself the result of a Φ-function, then availability transitively depends on availability at the Φ-function $h_j \leftarrow \Phi(\ldots)$ — it would be false if availability is false at the Φ-function of h_j. Finally, availability is true if $has\_real\_use_{h_j}$ = true for each Φ-operand h_j.

The algorithm to compute availability consists of two passes as shown in Fig. 4.7. First one is Reset_availability and next one is computation of availability at entry and exit of the nodes. Reset_availability takes up the value of *available* flag computed during Renaming stage and propagates false values to the corresponding uses in a Φ-function. For all Φ-functions $h_j \leftarrow (\ldots)$ with *available* value false, it resets the availability of all Φ-functions having h_j as a Φ-operand and $has\_real\_use_{h_j}$ as false. This propagation of boundary values of availability is performed recursively.

The second pass of the algorithm makes a preorder traversal on the dominator tree of the program flow graph to compute availability at the entry and exit of the node. We use Lemma 2 to compute availability at the entry of a node. Since the algorithm makes a preorder traversal on the dominator tree of the PFG, a node is visited only after all its dominators have been visited. If a node b_i does not contain a Φ-function, then *avail_at_entry* is simply *avail_at_exit* of its immediate dominator. If b_i contains a Φ-function then the availability at its entry is same as the value of *available* flag for the Φ-function.

4.3.1 Example

Figure 4.2 shows the dominator tree for the program of Fig. 4.1. The special version $\perp$ is assigned to $a * b$ at the exit of blocks b_1, b_5 and b_8 because $a * b$ is not available at their exit (see figure 4.6). Hence *available* value of all Φ-functions in the program $available_{h_2}$ = $available_{h_3}$ = $available_{h_4}$ = false. This is the value computed during renaming stage. During availability computation *avail_at_exit* becomes false for all nodes in G except for nodes b_2 and b_6. Note that $has\_real\_use_{h_1}$ = true for the Φ-function of h_4 located in node b_9 and $has\_real\_use_{h_2}$ = true for the Φ-function of h_3 located in node b_7. The result is presented in a tabular form.

The result is shown as tabular form. In the following tables, T means TRUE and F means FALSE.

```
Algorithm Compute_availability(Droot)
{
    $\forall$ $\Phi$-functions $h_j \leftarrow \Phi(\ldots) \ni$ $available[h_j]$ = false
        $\forall$ $\Phi$-functions $h_i \leftarrow \Phi(\ldots, h_j, \ldots) \ni$
        $available[h_i]$ = true and $has\_real\_use[h_j]$
        = false at $h_i$
            Reset_availability($h_i$);
    Let $x$ be the current node visited in preorder;
    If $x$ is the entry node of PFG, then
        $avail\text{-}at\text{-}entry[x] \leftarrow$ false;
    $\forall$ expressions $e$
        If ($x$ contains a $\Phi$-function) then {
            Let $h_i$ be the target of $\Phi$-function;
            $avail\text{-}at\text{-}entry[x] \leftarrow available[h_i]$;
        }
        Else {
            Let $Idom_x$ be Immediate dominator of $x$;
            $avail\text{-}at\text{-}entry[x] \leftarrow avail\text{-}at\text{-}exit[Idom_x]$;
        }
        If ($node\_type[x]$ is $empty$) then
            $avail\text{-}at\text{-}exit[x] \leftarrow avail\text{-}at\text{-}entry[x]$;
        Else If (($node\_type[x]$ is $avail$)
            or ($node\_type[x]$ is $both$)) then
                $avail\text{-}at\text{-}exit[x] \leftarrow$ true;
            Else $avail\text{-}at\text{-}exit[x] \leftarrow$ false;
}
Procedure Reset_availability($h_i$)
{
    $available[h_i] \leftarrow$ false;
        $\forall$ $h_k \ni h_k \leftarrow \Phi(\ldots, h_i, \ldots)$
            If $available[h_k]$ = true and $has\_real\_use_{h_i}$
            = false at $h_k$ then
                Reset_availability($h_k$);
}
```

Figure 4.7: Computation of availability

Nodes	1	2	3	4	5	6	7	8	9
avail-at-entry	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*
avail-at-exit	*F*	*T*	*F*	*F*	*F*	*T*	*F*	*F*	*F*

4.3.2 Analysis

The above algorithm to compute availability is linear with respect to the number of basic blocks in the PFG since it makes a single preorder traversal of the dominator tree and computes availability at the entry and exit of all nodes with respect to all expressions in the program. The Reset_availability module makes a worst case single pass on all nodes. But this is quite unlikely as all the nodes in a program very rarely have Φ-functions. Below we are presenting the correctness proof.

Lemma 3 Computation of availability algorithm computes availability correctly.
Proof : The computation of availability at the points containing Φ-functions is crucial to the correctness. An expression is not available at a Φ-function if the Φ-function contains a $\perp$ operand. During renaming, the *available*, which is equivalent to the *avail-at-entry* flag is made false. Also if a Φ-function, that has no $\perp$ operand has an operand that is defined by another Φ-function whose availability is false, then availability of the expression is false at that point. Such resetting of availability is performed by the *Reset_availability* pass. The availability at entry points with no Φ is correctly computed as proved by lemma 2. The propagation of availability from entry to exit point in a node is obvious. □

4.4 Computation of Anticipatability

Computation of anticipatability requires two passes. The first pass computes anticipatability at each Φ-function. This pass is analogous to the DownSafety pass of SSAPRE [C$^+$97] and the Reset_availability pass of our computation of availability algorithm. The second pass computes anticipatability at the entry and exit of each node. This is achieved through a postorder traversal of the dominator tree. The exit property of a node is propagated to its entry using the *node_type* attribute computed during Renaming. Anticipatability at the exit of a node x is computed from anticipatability at the entry of successor nodes. If anticipatability at the entry of any of the successors is not false, we recursively compute it before resuming the normal postorder traversal. If a successor is a loop entry node y then the *down_safe* value of the Φ-function in y is used. If we would have computed *ant-at-entry* of y recursively, then the algorithm would have worst case time complexity as nonlinear and/or the algorithm may give erroneous results. This is the reason why during Φ-insertion Φ's for all expressions are inserted at the entry of all loop-entry nodes. This technique also allows handling of any type of program flow graph. Figure 4.8 contains the algorithm for computation of anticipatability.

The data structures used in the algorithms are the two binary flags *avail-at-entry* and *avail-at-exit*. These two indicate the availability at the entry and exit of a node with respect to an expression respectively.

4.4.1 Example

In the PRESSA form of Fig. 4.6, down_safe is computed at nodes b_9, b_7 and b_3 during Renaming and DownSafety. It is *false* in the first case but *true* in the other two cases. The computation of anticipatability starts from node b_2 (See dominator tree in figure 4.2). Its successor b_9 is unvisited and has anticipatability at its entry as false. So *ant-at-exit* of b_2 is made false. Next the computation is done for b_4. It successor b_7 is unvisited and its *ant-at-entry* is true. Therefore the Anticipatability module is called recursively to compute anticipatability for the sub-dom-tree rooted at node b_7. b_3 is a loop entry node and therefore while computing *ant-at-entry* of b_7, no recursion is made for b_3. Anticipatability at entry of node b_8 is true since the node is *antloc*. Propagation of these values yields Anticipatability = *true* at the entry of blocks b_3, b_4, b_5, b_6, b_7 and b_8. The result of computation of anticipatability is tabulated as under.

Nodes	1	2	3	4	5	6	7	8	9
ant-at-entry	*F*	*F*	*T*	*T*	*T*	*T*	*T*	*T*	*F*
ant-at-exit	*F*	*F*	*T*	*T*	*T*	*T*	*T*	*F*	*F*

4.4.2 Analysis

This algorithm for computing anticipatability is a linear algorithm as each node is visited only once. When the computation of anticipatability at the exit of a node x is suspended and the algorithm is applied on a subtree rooted at an unvisited successor z of x, then all nodes which are dominated by z are visited in the process and their ant-at-exit and ant-at-entry are computed, Therefore the algorithm does not visit the nodes in such a subtree again and thus it is ensured that all nodes are visited only once. So the algorithm is of linear time complexity with respect to the number of nodes in the program flow graph.

```
Algorithm Compute_Anticipatability(Droot)
{
    $\forall x$
        $\textit{ant-at-exit}[x] \leftarrow \textit{ant-at-entry}[x] \leftarrow$ true;
    $\textit{ant-at-exit}[\textit{exit}] \leftarrow$ false;
    $\textit{ant-at-entry}[\textit{Droot}] \leftarrow$ Anticipatability($\textit{Droot}$);
}
boolean Anticipatability(Droot)
{
    Let $x$ be the current node in dom-tree visited in postorder;
    If ($x$ is already visited) then Return($\textit{ant-at-entry}[x]$);
    For all expressions $e$ do {
        If ($x$ is not exit) then {
            If ($\exists\ y \in$ succ($x$) $\ni$ $\textit{ant-at-entry}[y]$ = false) then
                $\textit{ant-at-exit}[x] \leftarrow$ false;
            Else {
                $\forall\ y\ \in\ succ(x) \ni\ y$ is not visited
                    Ant_at_Successors($y$);
                $\textit{ant-at-exit}[x] \leftarrow \Pi_{y \in succ(x)}$ ant-at-entry[$y$];
            }
        }
        If ($\textit{ant-at-entry}[x]$ != false)
            If ($\textit{node\_type}[x]$ = $\textit{empty}$) then
                $\textit{ant-at-entry}[x] \leftarrow \textit{ant-at-exit}[x]$;
            Else
                If ($\textit{node\_type}[x]$ = $\textit{avail}$ or $\textit{others}$) then
                $\textit{ant-at-entry}[x] \leftarrow$ false;
    }
    Return($\textit{ant-at-entry}[\textit{Droot}]$);
}
Procedure Ant_at_Successors($y$)
{
        If ($y$ is a loop entry node) then
            $\textit{ant-at-exit}[x] \leftarrow \textit{ant-at-exit}[x]$ and
                down_safe($\Phi$-function in $y$);
        Else
            $\textit{ant-at-exit}[x] \leftarrow \textit{ant-at-exit}[x]$
                and Anticipatability($y$);
}
```

Figure 4.8: Computation of anticipatability

Lemma 4 Computation of anticipatability algorithm computes anticipatability correctly.

Proof : The computation of anticipatability at the points containing Φ-functions is crucial to the correctness. An expression is not anticipatable at a Φ-function if the Φ-function is on top of renaming stack and a new version is being given to an original occurrence or exit point of the program is reached or a $\perp$ version is being given to the Φ-operand. These cases are taken care of during renaming. In no other cases except the Φ-function having a use as an operand in a Φ-function having anticipatability as false, anticipatability at a Φ-function can be false. Therefore after *DownSafety*, correct anticipatabilities have been computed at points containing Φ-functions. Since loop entry nodes also contain Φ-functions, the anticipatability at the entry of loop entry nodes are correctly known. Computing this property at the exit of a node is correct for obvious reason — anticipatability is a backward problem and is an all-path problem. Also propagation of anticipatability from exit of a node to its entry if not known, is apparently correct. □

4.5 Computation of Eliminatable paths

This step constructs an *eliminatability graph* G_l such that each path in G_l is an e-path in G with respect to a specific expression e in the program. This step is applied to the program expression by expression. Nodes and edges in G_l are the nodes and edges in G, except for one difference — some nodes of G are split into two nodes of G_l. This is done to facilitate construction of G_l. Node splitting is motivated by the following considerations: An e-path for expression e starts with a node whose *node_type* is *avail* or *both* (we will call this a *start node*), ends with a node whose *node_type* is *antloc* or *both* (an *end node*), and all intermediate nodes x in it have $node\_type_x = empty$ and $avail\_at\_exit_x = true$ or $ant\_at\_exit_x = true$. It may be noted that a node x with $node\_type = both$ could be the end of one e-path and start of another e-path. To simplify the identification of e-paths, we split each such node x into nodes x^{in} and x^{out}, which represent the parts containing the entry and exit of node x, respectively. All in-edges of node x become in-edges of x^{in} and all out-edges of node x become out-edges of x^{out}. Nodes x^{in} and x^{out} do not have any out-edges and in-edges, respectively. Figure 4.9 shows the splitting of a node. As described at the start of this section, the motivation for building G_l is to avoid having to trace individual e-paths in G.

The algorithm performs a preorder traversal on the dominator tree of the program flow graph and selectively adds nodes and edges to G_l. It performs the following actions for each node x visited during the traversal: If x can be an intermediate or end node of an

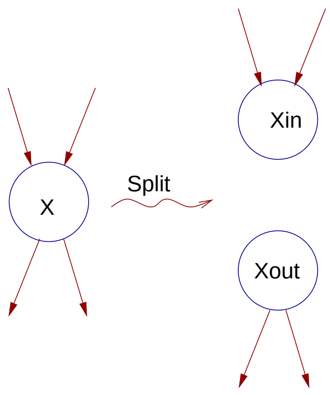

Figure 4.9: Illustration of node splitting

e-path (indicated by $node\_type_x$ = $empty/antloc/both$) and a predecessor p of x can be a start or intermediate node of an e-path (indicated by $node\_type_p$ = $empty/avail/both$) then edge (p, x) is added to G_l. Similarly an edge (x, s) is added for a successor s if $node\_type_x$ = $empty/avail/both$, and $node\_type_s$ = $empty/antloc/both$. It now splits x if $node\_type_x$ = $both$ as described earlier. This procedure is conservative, hence some nodes created in G_l may not belong to an e-path. Hence we prune the graph by removing all useless nodes from G_l, where a useless node is an isolated node or a node which has no successors but cannot be the end-node of an e-path or has no predecessors but cannot be a start node of an e-path. The removal of such node is important as we want a graph that contains only e-paths of the PFG and no other nodes that cannot be part of any e-path. After this step, the graph contains nodes which can be classified into start nodes, end nodes and intermediate nodes strictly according to Definitions of eliminatable occurrence and e-path. Note that G_l can be a multiple-entry multiple-exit graph which may contain one or more connected components. Figure 4.10 shows this algorithm. Note that a node x_l in G_l is a node that corresponds to node x in G. We use the superscript *in* or *out* if the node has been split while constructing G_l. It should be noted that a loop can exist in an e-graph if the loop in the e-graph is an empty loop-path with respect to the expression for which the e-graph is computed. Therefore if there exists a loop in an e-graph then there exists an empty loop path with respect to the expression in question in the corresponding program flow graph.

The *Append_node_to_paths* and *Append_successors* modules in the algorithm (see figure 4.10) check if a predecessor and a successor of the current node x can be a part of the e-graph respectively. If a predecessor and/or a node can be a part of an e-graph and already been visited, then the *add_edge* module is called. Modules $Remove\_Uselessnodes_p$

and *Remove_Uselessnodes*$_s$ are called recursively to remove nodes that cannot be part of any e-path, but still are in the computed e-graph. The *Remove_Uselessnodes*$_p$ module removes useless nodes having no predecessor and cannot be a start node in the e-graph. Similarly *Remove_Uselessnodes*$_s$ removes useless nodes having no successors and cannot be an end node in the e-graph.

Lemma 5 $x_l \dots z_l$ is a maximal path in G_l iff $x \dots z$ is an e-path in G.

Proof : *(a) If part:* Since x contains an available-at-exit occurrence, z contains a locally available occurrence and all intermediate nodes are empty and e is safe at their exit (see Defs. DPH–1 and DPH–2), nodes on the path $\alpha \equiv [x \ \dots \ z]$ will be added to G_l and none of them will be removed as a useless node. Therefore there will be a corresponding path $\alpha_l \equiv [x_l \dots z_l]$ in G_l. Since x and y are start node and end node, respectively, $|\text{pred}(x_l)| = 0$ and $|\text{succ}(y_l)| = 0$. Hence $[x_l \dots z_l]$ is a maximal path in G_l.

(b) Only if part: $\alpha_l \equiv [x_l \dots z_l]$ is a maximal path in G_l. An edge (p_l, q_l) is added to G_l only if an edge (p, q) exists in G. Hence there exists $\alpha \equiv [x \dots z]$ in G such that x is a start node and z is an end node. Let α not be an e-path in G. Hence some node b in it is not empty or e is not safe at its exit, or both. For such a node no node will be created in G_l, a contradiction. □

4.5.1 Example

Figure 4.11 illustrates the e-graph G_l for the program of Fig. 4.6. Node b_6 is split into nodes $b_{6_l}^{in}$ and $b_{6_l}^{out}$ since $node\_type_{b_6} = both$. Nodes b_{7_l}, b_{3_l} and b_{4_l} are added to G_l because they are *empty*. Nodes b_{5_l} and b_{8_l} are added because they are *antloc*. Nodes b_{2_l} and b_{9_l} will not be added to the egraph. Note that the loop b_{3_l}-b_{4_l}-b_{7_l}-b_{3_l} is empty with respect to the expression $a * b$.

4.5.2 Analysis

The time complexity of the algorithm is linear with respect to the number of the nodes in the program flow graph, as each node is visited only once during the preorder traversal of the dominator tree. The above algorithm is independent of any intermediate form and can be used to compute the candidate elim-paths in linear time. Thus it can also be used in frameworks not using SSA. The correctness of the algorithm that the computed e-graph has a path p in it iff p is an e-path in the corresponding PFG with respect to the specific expression e follows from *lemma*-3.

```
Algorithm Compute_elim_paths(Droot)
{
    Traverse the dominator tree in preorder;
    Set $visited[x]$ of current node $x$ to true;
    If (($x$ is neither entry nor exit node) and ($node\_type[x] = both$)) then
        Mark node $x$ for splitting;
    If ($node\_type[x] = empty$) and ($avail\text{-}at\text{-}exit[x]$ or $ant\text{-}at\text{-}exit[x]$) then {
        Append_node_to_paths($x$);
        Append_successors($x$);
    }
    Else If ($node\_type[x] = antloc$) then Append_node_to_paths($x$);
        Else If ($node\_type[x] = avail$) then Append_successors($x$);
            Else If ($node\_type[x] = both$) then {
                Append_node_to_paths($x$);
                Append_successors($x$);
            }
    Uselessnodes$_p$ $\leftarrow \{x_l \in G_l \mid node\_type_x \neq (both \text{ or } avail) \text{ and } |\text{pred}(x_l)| = 0\ \}$
    Uselessnodes$_s$ $\leftarrow \{x_l \in G_l \mid node\_type_x \neq (both \text{ or } antloc) \text{ and } |\text{succ}(x_l)| = 0\ \}$
    $\forall x \in$ Uselessnodes$_p$ Remove_Uselessnodes$_p(x)$;
    $\forall x \in$ Uselessnodes$_s$ Remove_Uselessnodes$_s(x)$;
}
Procedure Append_node_to_paths($x$)
{
    $\forall\ y \in$ pred($x$) $\ni$ ($visited$[y] and $node\_type_y$=$empty/avail/both$) Add_edge($y, x$);
}
Procedure Append_successors($x$)
{
    $\forall y \in$ succ($x$) $\ni$($visited$[y] and $node\_type_y$=$empty/antloc/both$) Add_edge($x, y$);
}
Procedure Add_edge($x, y$)
{
    If ($x$ has been marked for splitting) then Create node $x_l^{out}$ in $G_l$ if not present;
    Else create node $x_l$ in $G_l$ if not present;
    If ($y$ has been marked for splitting) then Create node $y_l^{in}$ in $G_l$ if not present;
    Else create node $y_l$ in $G_l$ if not present;
    Add a directed edge ($x_l^*, y_l^{**}$) where '*' is 'out', or blank and '**' is 'in' or blank;
}
Procedure Remove_Uselessnodes$_p(y)$
{
    $S_y \leftarrow$ succ($y$); Remove $y$ from $G_l$;
    $\forall z \in S_y \ni |\text{pred}(z)| = 0$ Remove_Uselessnodes$_p(z)$;
}
Procedure Remove_Uselessnodes$_s(y)$
{
    $S_y \leftarrow$ pred($y$); Remove $y$ from $G_l$;
    $\forall z \in S_y \ni |\text{succ}(z)| = 0$ Remove_Uselessnodes$_s(z)$;
}
```

Figure 4.10: Computation of e-paths

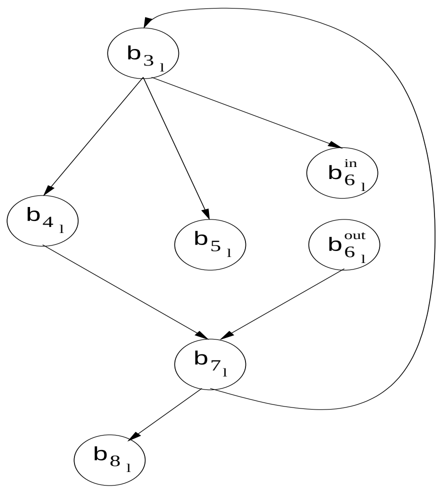

Figure 4.11: Illustration of e-path

4.6 Computation of Insertion Points

To perform insertion according to the code placement criteria DPH–1 and DPH–2, we need to identify all nodes y such that edge (y, x) exists in G where x is an intermediate node or end node of an e-path and edge (y, x) does not belong to an e-path. Placement will now be performed in edge (y, x) or in node y depending on part (c) of these criteria. To implement the placement we identify all edges (y, x) in G such that x is not the start node of an e-path, edge (y, x) does not exist in G_l and the expression is not available at the exit of y in G, and mark them as potential insertion points. The decision whether to insert in an edge (y, x) or in node y is taken during insertion in the next step. The marked edges are held by a set *Insert*. Figure 4.12 shows the algorithm. When applied to the G_l of Fig. 4.11, this algorithm computes Insert = $\{(b_1, b_3),\ (b_5, b_7)\}$.

4.6.1 Analysis

The above algorithm is linear with respect to the number of the nodes in the original program flow graph. In fact, due to the for loop on the number of incoming edges to find insertion points, the time complexity is $O(e)$, where e is the number of edges in the PFG. This algorithm is also independent of any intermediate representation and can be used to insertion points provided. This algorithm is very simple and straight forward in its technique.

Lemma 6 Computation of insertion points algorithm computes all and correct insertion

```
Algorithm Compute_Insertion_Points(E-graph $G_l$)
{
    Insert = {};
    $\forall\ x_l\ \in\ G_l$
        If | pred($x_l$) | $\neq 0$ then
            $\forall$ edges $(y, x) \in\ G \ni ((y_l, x_l) \notin G_l$ and $avail\_at\_exit[y]$ = false)
                Insert = Insert $\bigcup$ $\{(y, x)\}$;
}
```

Figure 4.12: Finding insertion points

points.

Proof : The correctness follows from the placement criteria DPH–1 and DPH–2 (see section 2.3.1). $\square$

4.7 Insertion

The insertion step handles three issues: Deciding whether insertions should be made in nodes or along edges, determining the correct SSA versions of expressions to be inserted, and handling extraneous Φ-functions. The previous step has marked edges (y, x) for insertion where x is a node in some e-path $w \ldots x \ldots z$ and edge (y, x) does not belong to an e-path. Part (c) of the placement criteria DPH–1 and DPH–2 dictate that insertion should be performed in node y if all paths starting on y have a prefix $y \ldots z$ such that node z contains an eliminatable occurrence of e, else insertion should be along an edge (y, x). We implement this by inserting an occurrence of e in node y if all its out-edges are marked for insertion, else we perform insertion only in the marked edges by splitting each marked edge to introduce a synthetic block [Dha88a]. This technique of insertion minimizes the number of computations inserted in the optimized program. We maintain renaming stacks for all operands of an expression, as also its result name, so that a correct SSA version of the expression can be constructed for insertion.

Some Φ-functions for e may be extraneous as follows: Consider a Φ-function $h_l \leftarrow \Phi(\ldots)$ which is followed by an occurrence of h_l along some path. If this occurrence of h_l is eliminatable, then an appropriate version of t, say t_i, would be allocated to h_l and the Φ-function would be replaced by $t_i \leftarrow \phi(\ldots)$ in the next step. If no occurrence of h_l is eliminatable, then the Φ-function for h_l is extraneous. Such a function should be removed to control the size of the SSA graph. Hence we detect an extraneous Φ-function and give a new version number, say h_k, to the first occurrence $h_l \leftarrow \ldots$ along a path starting on the Φ-function. Uses of h_l dominated by this version are now replaced by h_k. The extraneous Φ-function is marked for deletion in the next step. When an edge (y, x) is marked for insertion of e, node x already contains a Φ-function for e. This function has a Φ-operand corresponding to the path ending in edge (y, x). When an expression $h_g \leftarrow e$ is inserted in node y, or along edge (y, x), the name h_g replaces the original Φ-operand in the Φ-function situated in x. The data structure used for marking extraneous Φ-functions is the binary flag *extraneousPhi*. Initially it is false for every Φ-function. At the end of insertion, it contains true value for the Φ-functions that are extraneous.

The module *Collect_versions* scans through all the instructions in the basicblock x and collects the versions of the variables as well as the expression variable h^e. Although in another module (see figure 4.13), to detect extraneous Φ-functions, actually detection of Φ-functions are done during collection of versions as all the statements are scanned by this module. Figure 4.13 contains the algorithm for insertion.

The *Detect_ExtraneousPhi* module detects the extraneous Φ-functions and its uses are given versions as mentioned as above. The algorithm does not mention how to process the uses. The module of *Detect_ExtraneousPhi* can be removed from the algorithms and the *Collect_versions* module can be changed to do its task. this. The algorithm in figure 4.14 detects extraneous Φ-functions and processes its uses along with collection of versions (in module *Collect_versions*). The call of module *Detect_ExtraneousPhi* has to be removed from module *Insert* in figure 4.13.

4.7.1 Example

Fig. 4.15 illustrates insertions performed by this algorithm. Insertions have been performed in node b_5 and along edge b_{1-3}. The algorithm checks that the single outgoing edge from node b_5 is in the set *Insert* and therefore the insertion has to be done at the exit of the node b_5. The SSA versions assigned to these insertions replace the $\perp$ operands in the Φ-functions situated in nodes b_7 and b_3, respectively. The Φ-function in node b_9 is marked as an extraneous Φ-function as it does not have any use in the program.

4.7.2 Analysis

The above algorithm is a linear order algorithm with respect to the number of nodes in the original program flow graph. It makes a preorder traversal on the dominator tree. The complexity is $O(e)$, where e is the number of edges in the program flow graph. This

```
Algorithm Insert(Droot : root of dom-tree)
{
    Let $x$ be the current node of dom-tree in preorder;
    Collect_versions($x$);
    If $\{(x, s) \mid s \in \mathrm{succ}(x)\} \subseteq$ Insert then Insert_in($x$);
    Else $\forall\ (x, s) \ni s \in \mathrm{succ}(x)$
        If ($(x, s) \in$ Insert) then {
            Insert a synthetic block $b_{x-s}$ to split the edge $(x, s)$;
            Insert_In($b_{x-s}$);
        }
    Detect_ExtraneousPhi($x$);
    If all children of $x$ in dom-tree have been visited then
        Pop all the versions pushed onto the
        renaming stacks due to statements in $x$;
}
Procedure Insert_In($z$)
{
    Assign a new version, say $h_k$, to the target $h$ of inserted computation;
    Push the $h_k$ onto the renaming stack of $h$;
    Replace the corresponding $\Phi$-operand (if any) in the successor(s) of $z$ by $h_k$;
    Insert an SSA version of the assignment $h_k \leftarrow e$ at the exit of $z$ using
        versions of the operands at the top of their renaming stacks;
}
Procedure Detect_ExtraneousPhi($x$)
{
    If (($x$ is not in an e-path) and ($x$ contains a $\Phi$-function $h_i \leftarrow \Phi(\ldots)$)
    and ($\mathit{avail\text{-}at\text{-}entry}[x]$ = false or $h_i$ has no use in the program)) then
        $\mathit{extraneousPhi}[x] \leftarrow$ true;
    If $\exists$ an occurrence of $e$ with version $h_k$ in $x \ni$
    an extraneous $\Phi$-function (of some node) has the same version number $h_k$ then
        Assign a new version to $h_k$;
        Push it onto the renaming stack of $e$;
        Replace all other uses of $h_k$ that are dominated by the
            occurrence of $e$ with version $h_k$ by this new version;
}
Procedure Collect_version($x$)
{
    $\forall$ assignments of the form $h \leftarrow \ldots$
    or an assignment to an operand $\mathit{opd}$ of $e$ in $x$
    in the order of their lexical occurrences in $x$
        Push the version given to $h$ (or $\mathit{opd}$)
        onto the renaming stack of $e$ (or $\mathit{opd}$);
}
```

Figure 4.13: Insertion of expressions

```
Procedure Collect_version($x$)
{
    $\forall$ statements in lexical order in $x$ do {
        If it is an assignment of the form $h \leftarrow \ldots$
        or an assignment to an operand *opd* of $e$ in $x$
            Push the version given to $h$ (or *opd*)
            onto the renaming stack of $e$ (or *opd*);
        If (($x$ is not in an e-path) and (the statement is a $\Phi$-function $h_i \leftarrow \Phi(\ldots)$
        and (*avail-at-entry*[$x$] = false or $h_i$ has no use in the program)) then
            *extraneousPhi*[$x$] $\leftarrow$ true;
        If the statement is an original occurrence with $h_i$
        and definition of $h_i$ is an extraneous $\Phi$-function, then
            If the $h$ version at the top of expression renaming stack is
            same as $h_i$, then {
                Assign a unique version to this statement;
                Push the version onto the expression renaming stack.
            }
            Else
                Change version $h_i$ of this statement to a unique version $h_k$.
        Else
            If the version $h_g$ at the top of expression renaming stack is not
            same as the version $h_i$ of this statement then
                Push this version $h_i$ onto the expression renaming stack.
    At the exit of node $x$ do {
        For each successor $y$ of $x$ do {
            If $y$ contains a $\Phi$-function and the definition of the $\Phi$-operand
            corresponding to $x$ is an extraneous $\Phi$-function $h_k \leftarrow \Phi(\ldots)$ then
                If version at the top of expression renaming stack is same as $h_k$ then
                    Replace the operand by $\perp$;
                Else Replace the operand version by the version at the top of the
                    expression renaming stack.
        }
    }
}
```

Figure 4.14: Algorithm for collecting versions, detecting extraneous Φ-functions and processing its uses. This is to be used by the Insertion algorithm of figure 4.13.

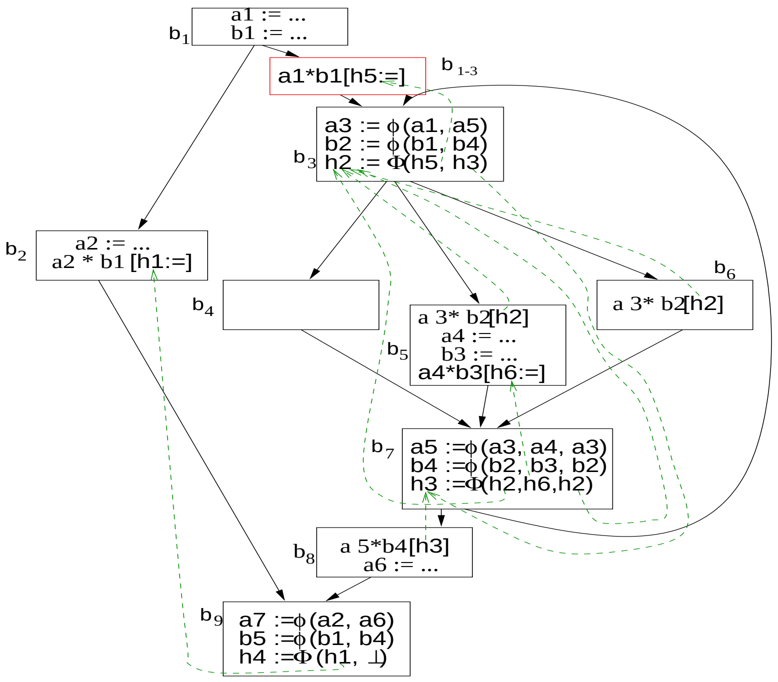

Figure 4.15: The PFG after insertion of computations.

is because, while reassigning versions to Φ-operands in successor, in the worst case the algorithm has to visit all the successors of a node. But since the number of edges is linear in the number of nodes, the complexity can be rewritten as $O(n)$. The logic of this algorithm is very simple and makes fullest use of the advantages of eliminatable paths. The correctness of the algorithm lies in the correctness of computation of insertion points and the versions of the operands in the inserted versions. The collection of versions maintains the renaming stacks properly. The correctness of processing of uses of extraneous Φ's is straight-forward from the algorithm itself (see figures 4.14 and 4.13).

4.8 Elimination of Redundant Computations

This step is the final step in PRE and its output is the PRE-optimized program with respect to a specific expression. This step first removes all extraneous Φ-functions marked in the previous step. It then traverses the SSA graph of h to assign a new version of temporary t, say t_i, to every version of h, say h_l. The definition $h_l \leftarrow e$ in the start node of an e-path and in a node marked for insertion is replaced by $t_i \leftarrow e$. Each occurrence $h_k \leftarrow \Phi(\ldots h_l \ldots)$ is replaced by $t_g \leftarrow \phi(\ldots t_i \ldots)$, where t_g is a unique temporary name, and each use of h_k

is replaced by t_g. If $t_g \leftarrow \phi(\ldots)$ contains an h version, say h_m, as an operand, then h_m is replaced by a unique t_n in the definition and all uses of h_m. The algorithm is given in Fig. 4.16.

4.8.1 Example

Figure 4.17 illustrates the optimized program after this step. The definition $h_6 \leftarrow a * b$ in node b_5 has been replaced by $t_4 \leftarrow a_4 * b_3$ and the occurrence of h_6 in b_7 is replaced by t_4 (and the Φ-function is replaced by a ϕ-function). The name t_1 has been used for the insertion $h_5 \leftarrow a * b$ in node b_{1-3}. The occurrence of h_5 in the Φ-function $h_2 \leftarrow \Phi(h_5, h_3)$ of node b_3 leads to replacement of the Φ-function by the ϕ-function $t_2 \leftarrow \phi(t_1, h_3)$ and replacement of h_2 in b_5, b_6 and b_7 by t_2. It also leads to replacement of h_3 by t_3 in nodes b_3, b_7 and b_8.

4.8.2 Analysis

The algorithm is of linear time complexity with respect to the number of the nodes in the program flow graph. In fact it is linear in terms of the number of edges in the program flow graph as while traversing the def-use chains, in worst case the algorithm has to traverse all its predecessors and successors, unless they have already been visited through another real occurrence.

4.9 Implementation of E-path_PRE

We have implemented the algorithms of E-path_PRE. Using this prototype, we have tested our algorithms for a number of specific test cases. Appendix A presents a detailed discussion on the interface, front-end and the features of the prototype along with some results. The prototype computes the SSA and PRESSA form and using it as input, applies our E-path_PRE algorithm.

```
Algorithm Eliminate_redundancy(PFG G)
{
    Remove all extraneous $\Phi$-functions;
    Traverse each SSA def-use graph;
    Let node $x$ contain a definition $h_i \leftarrow \ldots$
    If (the definition is $h_l \leftarrow \Phi(\ldots)$) then
        If ($x$ is an intermediate or end node) then {
            Generate a unique $t_i$ for $h_l$;
            Replace the $\Phi$-function by $t_i \leftarrow \phi(\ldots)$;
            Replace_use($h_l, t_i$);
            If ($t_i \leftarrow \phi(\ldots)$ has an $h$ operand) then Replace_operand($t_i$);
        }
    Else If ((the definition of $h_l$ is an inserted occurrence or $x$ is a start node) or ($x$ has a
        definition and a real occurrence of $h_l$)) then {
            Generate a unique $t_i$ for $h_l$;
            Replace the definition $h_l \leftarrow e$ by $t_i \leftarrow e$;
            Replace_use($h_l, t_i$);
        }
    Remove all remaining $\Phi$-functions;
    Remove all remaining $h$ versions, without removing the occurrences of the expressions;
}
Procedure Replace_use($h_l, t_i$)
{
    For each entry in the def-use chain of $h_l$
        If use is of the form $h_k \leftarrow \Phi(\ldots h_l \ldots)$ then {
            Generate a unique name $t_g$ and replace
            the $\Phi$ function by $t_g \leftarrow \phi(\ldots t_i \ldots)$;
            Replace_use($h_k, t_g$);
            If ($t_g \leftarrow \phi(\ldots)$ contains an $h$ operand)
            then Replace_operand($t_g$);
        } Else Replace the use by $t_i$;
}
Procedure Replace_operand($t_j$)
{
    For each $h$-operand $h_k$ of $t_j \leftarrow \phi(\ldots)$ {
        Generate a unique $t_l$ for $h_k$;
        Replace the operand by $t_l$;
        If the definition of $h_k$ is $h_k \leftarrow \Phi(\ldots)$ then {
            Replace the $\Phi$-function by $t_l \leftarrow \phi(\ldots)$;
            Replace_use($h_k, t_l$);
            If ($\exists$ an $h$ operand of $t_l$) then Replace_operand($t_l$);
        } Else if the instruction is an original occurrence or an inserted one {
            Replace the definition $h_k \leftarrow e$ by $t_l \leftarrow e$;
            Replace_use($h_k, t_l$);
        }
    }
}
```

Figure 4.16: Elimination of redundancies

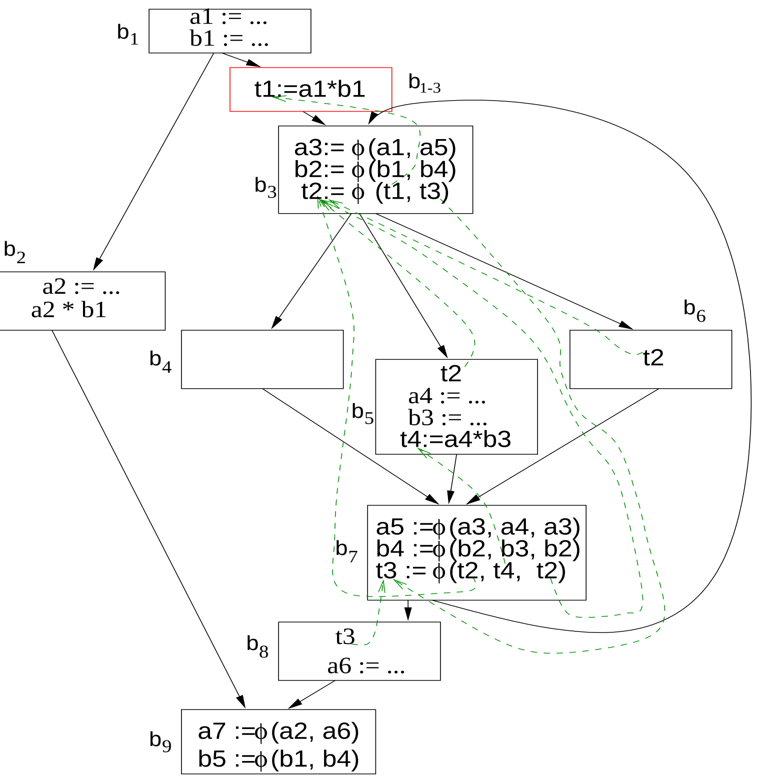

Figure 4.17: The optimized program

Chapter 5

Algorithms of E-path_SRLE

E-path_SRLE is an extension of E-path_PRE to perform strength reduction of large expression along with PRE. The most significant extension is instead of PRESSA form, E-path_SRLE uses SRLESSA form. All other stages are straight-forward and therefore we will treat them with less importance as compared to $\Phi_{1,2}$-insertion and Renaming. The SRLESSA form facilitates efficient computation of the srle, Xsrle and sr-properties. Once the properties are computed, the computation of egraph and insertions are analogous to those in E-path_PRE.

5.1 SRLE candidates and $\Phi_{1,2}$-Insertion

The reason behind inserting $\Phi_{1,2}$-functions in SRLESSA form is as follows. In SRLE, along a path, at most one update computation is to be inserted. Therefore for computing placement points for update computations, we need to compute srle-availability and srle-anticipatability and other srle and Xsrle-properties. Recalling the definition of srle-properties, these properties are computed by bypassing exactly one update definition in between the previous occurrence and current occurrence. For recomputation insertion, we need to compute the normal properties. We perform the computations of both normal properties and srle-properties simultaneously by using SRLESSA form and the $\Phi_{1,2}$-function. The $\Phi_{1,2}$-function is as under —

$(h, s) \leftarrow \Phi_2((h, s), \ldots, (h, s))$. The h in the pair (h, s), is for computing normal properties and s is for computing srle-properties (see section 1.3.2).

During $\Phi_{1,2}$-insertion, SRLE-candidate expressions are detected. If an expression is an SRLE-candidate, then Φ_2-function is inserted for it, otherwise Φ_1-function is inserted. The insertion points are same as that of Φ-functions of PRESSA. The use of Φ_2-functions is

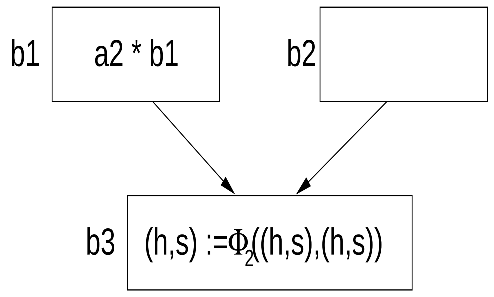

Figure 5.1: Illustration of Φ_2-function

essentially to compute srle-availability and srle-anticipatability.

5.1.1 Example

The figure 5.1 is an example showing the Φ_2-example. Let the expression $a*b$ be a candidate for SRLE. Therefore a Φ_2-function has been inserted in node b_3. The Φ_2-function is as under — $(h^{a*b}, s^{a*b}) \leftarrow \Phi_2((h^{a*b}, s^{a*b}), (h^{a*b}, s^{a*b}))$.
The first (h^{a*b}, s^{a*b}) is for the predecessor b_1 of b_3 and the next is one for b_2.

Figure 1.2 is the example for illustration of SRLE. We have chosen a simple example as here our main objective is to illustrate our technique of strength reduction only. Figure 5.2 is the dominator tree for this example. In figure 1.2, there are two expressions $i * k$ and $j * l$. Both of them have an operator $*$ and therefore are candidates for SRLE. Therefore for both of them Φ_2-functions are inserted. For the expression $i * k$, the Φ_2-function inserted in nodes b_2 and b_5 is

$(h^{i*k}, s^{i*k}) \leftarrow \Phi_2((h^{i*k}, s^{i*k}), (h^{i*k}, s^{i*k}))$.

and similarly for expression $j * l$, the Φ_2-function inserted in nodes b_2 and b_5 is

$(h^{j*l}, s^{j*l}) \leftarrow \Phi_2((h^{j*l}, s^{j*l}), (h^{j*l}, s^{j*l}))$.

Here the ordered pair of (h, s) is important. The first h in the pair is given version using the normal renaming as in PRESSA and the second h is given versions by bypassing exactly one update computation of an operand of the expression between the last occurrence and current occurrence. The second one is used therefore for computation of srle-properties whereas the first one is used for normal properties. But for expression $d + e$, Φ_1 is inserted in nodes b_2 and b_5 as it is not an SRLE-candidate expression. The Φ-function for $d + e$ inserted in nodes b_2 and b_5 is

$h^{d+e} \leftarrow \Phi_1(h^{d+e}, h^{d+e})$.

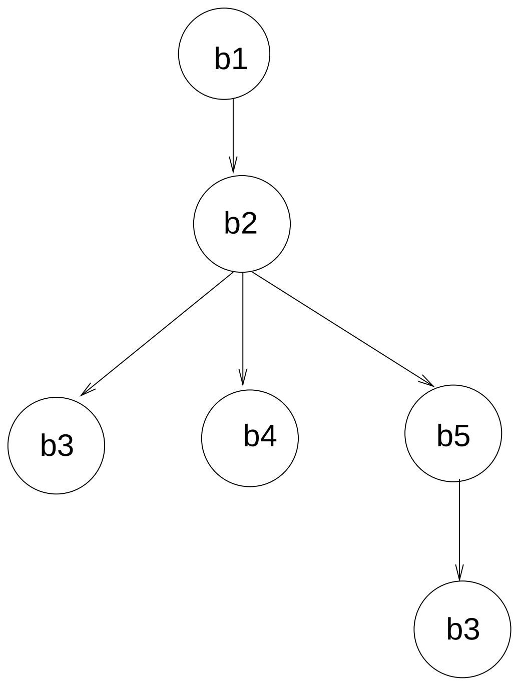

Figure 5.2: Dominator tree of the program of figure 1.2

5.2 Renaming

For non-SRLE candidates, the process of renaming, computation of availability and anticipatability remain same as that of PRESSA. For SRLE-candidate expressions, the renaming scheme is conceptually same, but the difference arises due to the paired operands of Φ_2-functions. Note that the s in the pair (h, s), is to compute srle-properties by bypassing exactly one update definition. Therefore while giving a version to the current occurrence of an SRLE-candidate expression e, it is checked if after the previous occurrence of e, and before the current occurrence, along a path p, there is an update definition of an operand. If there is such a definition, then assign the version of s of previous occurrence to the s in the pair (h, s).

Renaming of SRLE-candidate expressions is done as per the following :

For each SRLE-candidate there are two expression renaming stacks — one for h and the other for s of the pair (h, s). First assign version to h using the PRESSA technique. If the version given to the Φ_2-operand then then at least one of the h and s in the pair gets a $\perp$ version. While renaming if the h is not $\perp$ then assign $\perp$ to s of the pair. If h is given $\perp$, then for the operand, for which there is mismatch in version, check if it has an exactly one update definition and no redefinition between the last occurrence of e and this point. This can be done as follows — let the variable of mismatched versions is v_i. Let the top of expression renaming stack contains v_j. If the definition of v_i is as $v_i \leftarrow v_j + k$, where k is a region constant, then there is exactly a single update definition of v between the last occurrence and the current occurrence of e; otherwise either there is a redefinition or more than one update computations of v.

If there is a single update definition, then assign the current version of s (the second one of the (h, s) pair), *i.e.* the version at the top of renaming stack of second s^e to it else assign $\perp$ to it. The checking of whether a update definition is there or not can be done by following the use-def chain of the corresponding operand. This renaming scheme computes flags *available*, *down_safe* and *has_real_use* and the srle-counterparts[1] of *available*, *has_real_use* and *down_safe*. The possible values of *node_type* excluding the earlier values in E-path_PRE are *Xsrle-both*, *Xsrle-antloc*, *Xsrle-avail* and *Xsrle-empty*.

Computation of normal flags *available*, *down_safe* and *has_real_use* are computed during the renaming of h in (h, s) pair. Their srle-counterparts are computed during the renaming of s. The technique of computation of these flags are same as that in PRESSA.

5.2.1 Example

After renaming, the state of Φ_1 and Φ_2-functions are as under : In node b_2, the Φ_2-functions for $i * k$ and $j * l$ are respectively —

$(h_1^{i*k}, s_1^{i*k}) \leftarrow \Phi_2((\perp, \perp), (h_2^{i*k}, \perp))$.

and $(h_1^{j*l}, s_1^{j*l}) \leftarrow \Phi_2((\perp, \perp), (\perp, s_2^{j*l}))$.

The Φ_1-function for $d + e$ is $h_1^{d+e} \leftarrow \Phi_1(\perp, h_2^{d+e})$.

In node b_3, the the Φ_2-functions for $i * k$ and $j * l$ are respectively —

$(h_2^{i*k}, s_2^{i*k}) \leftarrow \Phi_2((h_1^{i*k}, \perp), (h_1^{i*k}, \perp))$.

and $(h_2^{j*l}, s_2^{j*l}) \leftarrow \Phi_2((h_1^{j*l}, \perp), (h_1^{j*l}, \perp))$.

The Φ_1-function for $d + e$ is $h_2^{d+e} \leftarrow \Phi_1(\perp, \perp)$.

The value of *srle-available* for nodes b_2 and b_5 are false for both $i * k$ and $j * l$. But *available* is true for these two expressions for node b_5. The value of *available* for nodes b_2 and b_5 are false. The value of *has_real_use* for expression $i * k$ and $j * l$ is true for node b_5 along its two predecessors b_3 and b_4. For expression $j * l$, the *has_real_use* value for node b_2 is false along predecessor b_5, but *srle-has_real_use* is true along this predecessor. The reason is the update definition of j in the node b_5. *has_real_use* is true for node b_2 along the predecessor b_5. All other possible values of *has_real_use* and *srle-has_real_use* are false.

[1]srle-counterpart of a flag or a property x is named as srle-x. The property name is prefixed by srle

5.3 Computation of Availability

Computation of availability is same as that in E-path_PRE except for the *node_type* values and the handling of s of the pair. It should be noted that at least one of the h and s in the pair has a $\perp$ version for a Φ_2-operand. Similarly availability and anticipatability for an expression are false for at least one of the h and s in the pair. Therefore for availability computation h of the (h, s) pair of the Φ_2 operand is used. If availability at a point is computed to be true, then srle-availability at that point is made false, otherwise for computation of srle-availability the flags srle-available and srle-has_real_use are used and the same technique as that of availability is followed. In this case s of the (h, s) pair of the operand is to be used.

Initially two passes are made to propagate the false value of available and srle-available. Reset_available and Reset_srle-available are used to perform this. Next a preorder traversal on the dominator tree is needed for computation of availability and srle-availability. Note that for computing two properties we are using a single traversal. After computing availability (srle-availability) at the entry of a node, it is propagated to the exit as in E-path_PRE. For srle counterpart the Xsrle-values of *node_type* are used; srle-availability at entry of a node gets propagated to exit if *node_type* is *Xsrle-empty*; srle-availability at exit is true if the value is *Xsrle-both* or *Xsrle-avail.* For all other values it is false. Also the *srle-availability at exit* is made true if *availability* at the entry is true, *srle-available* at that node entry is false and *node_type* is *Xsrle_empty.* If *srle-availability* is true at the entry of a node x and *node_type* value of x is *Xsrle_empty*, then make *srle-availability* at the exit of x as false.

Extension of computation of anticipatability for SRLE is analogous to the extension of computation of availability.

5.3.1 Example

For the example of figure 1.2, for expression $j * l$ the availability at the entry of node b_5 is true, and *srle-availability* is false. But *srle-availability* is true at the end of node b_5 as availability is true at its entry and *node_type* is *Xsrle-empty.*

5.4 Computation of e-paths

The computation of e-graph is applied expression by expression. This is same as that of PRESSA except that it is preceded by finding that whether an SRLE-candidate is a proper-SRLE-candidate. This can be done by looking at the $*$ or $**$ operators and the operands involving it. If an expression is not a proper candidate then the remaining four stages are same as those of E-path_PRE else stages for SRLE will be used. In computing e-graph of the PFG with respect to a proper SRLE-candidate e is done by using sr-safety instead of safety. The *node_type* values now are the sr-counterparts of the local values. As mentioned earlier, the sr-counterpart of a local property is the logical OR of the normal value and the Xsrle-value. Therefore the values of *node_type* are *sr-both*, *sr-empty*, *sr-avail*, *sr-antloc* and *others*. The algorithm 4.10 is used here also, with the exception that instead of the normal local values of *node_type* used in it, the corresponding sr-values are used. Useless nodes are also removed from the computed egraph as in E-path_PRE. During the preorder traversal while computing srle-egraph, it replaces the Φ_2-function of the expression by Φ-function. This is done to reduce the size of the SSA graph for the expression as SRLESSA form is no more needed for the expression. Note that SRLESSA form for an expression is needed to compute sr-safety and node_type values. So a Φ_2-function $(h, s^{'}) \leftarrow \Phi_2((h, s^{'}), \ldots)$ is replaced by $h \leftarrow \Phi(h, \ldots)$. Note that s of the pair is removed as the purpose of s is to compute srle-properties, that is already over.

5.4.1 Example

In the figure 1.2 the expression $i * k$ is a loop invariant expression. Therefore computation of egraph of E-path_PRE is applied to this expression. The egraph for expression $i * k$ is shown in figure 5.3. Node b_1 is not in the graph as it cannot be start nor intermediate nor an end node of an srle-epath. Note that the nodes b_3 and b_4 are splitted as both of them have *node_type* for $i * k$ as *both*.

The srle-egraph for expression $j * l$ is also same as the egraph of $i * k$ as $j * l$ is a proper candidate of SRLE and therefore the srle-egraph is computed for it. The Φ_2-functions for expressions $i * k$ and $j * l$ have been replaced by corresponding Φ_1-functions in nodes b_2 and node b_5. The egraph for expression $d + e$ is is of zero size as the expression is not partially redundant in the program.

5.4.2 Analysis

The stage computes srle-egraph and therefore the resultant graph contains paths which are srle-epaths. Since a single preorder traversal is taken, the algorithm is a linear one. The correctness of the algorithm follows from the correctness of the algorithm of computation of egraph of E-path_PRE.

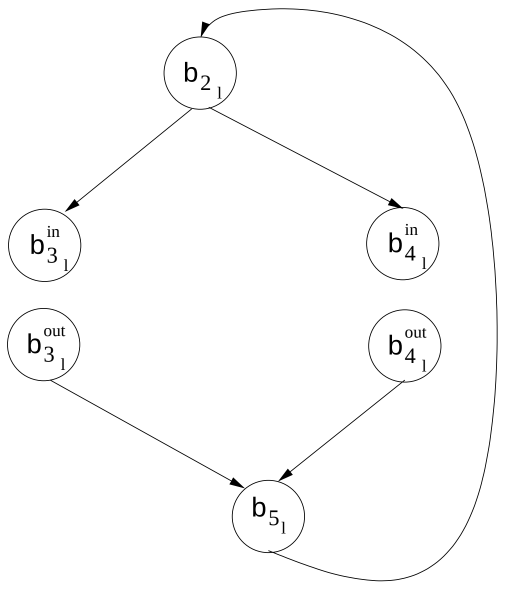

Figure 5.3: SRLE-egraph for expressions $i * k$ and $j * l$

5.5 Computation of Insertion points and Insertion

Insertion points for recomputations are computed same as done in E-path_PRE. Insertion points for update computations is done as under : An update computation is placed at the exit of a node j iff j is an intermediate node in e-graph and the expression e is srle-available at the exit of j. Insertion of recomputations is same as the insertion stage of E-path_PRE. In this stage recomputations are inserted.

The insertion points for update computations are either entry of a node or exit of a node. For finding such points a pass is made over all the nodes in the srle-egraph. The insertion points are then computed using the conditions in DPS-3 and DPS-4 (see section 2.3.1).

The insertion algorithm takes care of extraneous Φ-functions. Note that computation of (srle-)egraph and insertion points, insertion, strength reduction, if needed and elimination of redundancy are performed collectively an expression by expression basis. But for more clarification we have given all the insertion points and PFGs with insertions in this stage, whereas actually they got computed after SRLE/PRE of the previous computation.

5.5.1 Example

The insertion point for expression $i*k$ is edge (b_1, b_2). The insertion point for recomputation of expression $j*l$ is edge (b_1, b_2). The insertion points for update computations of expression

$j * l$ are exits of node b_5 and b_6. The insertion is done at the exit of node b_1 as (b_1, b_2) is the single outgoing edge of b_1 and therefore the insertion algorithm marks exit of b_1 as the exit node.

5.6 Insertion of Update Computations

In this stage update computations are added. A preorder traversal is performed on the dominator tree. The purpose of the preorder traversal is to collect the variable and expression variable versions. We do not hold the versions from last stage as we need to keep track of the versions at the entry and exit of each node for each variable and expression variable. This requires a large amount memory. At an insertion point an update computation for the expression is generated and its proper version is generated.

The stage of elimination of redundancy is same as that of E-path_PRE. The final program of the example after final application of elimination of redundancy and in standard SSA form as shown in figure 1.5(b).

Chapter 6

Concluding Remarks

6.1 Analysis of our Approach

The advantages of E-path_PRE and E-path_SRLE are simplicity, understandability and efficiency. E-path_PRE does not involve use of complex data flows. It uses only well-known fundamental data flows of available and anticipatable (i.e. very-busy) expressions [MR79, Muc97]. It also involves less work than SSAPRE, the SSA-based approach of [C$^+$97]. In the four stages which are applied on an expression by expression basis, E-path_PRE performs five passes over the program, which matches the number of passes in SSAPRE. Computation of e-paths requires two passes whereas WillBeAvail stage of SSAPRE also requires two passes. However, the work done during the e-paths computation passes is much less than the work done by SSAPRE during the WillBeAvail passes, since our passes are mere graph-related as against the WillBeAvail passes which need to process the computations in each node. Computation of insertion points is performed on the e-graph which is smaller in size compared to a PFG. E-path_PRE computes seven global properties, viz. available, down_safe, availability, anticipatability, extraneousPhi, has_real_use and Insert; and one local property node_type of a node. SSAPRE computes nine global properties, viz. has_real_use, down_safe, can_be_avail, will_be_avail, later, insert, avail_def, save and reload. In fact, in our approach since we are using the flag down_safe for computing anticipatability at the entry and exit points of nodes, the ant-at-entry flag can use the memory of down_safe flag as they are equivalent and down_safe is not needed later in the program. Similar technique can also be used for avail-at-entry and available flags. This effectively reduces the number global properties computed by our algorithm to five, a significant improvement as against SSAPRE.

As we have detected, SSAPRE incorrectly computes the anticipatability (or equivalently downsafety) at the program points containing the Φ-functions. Therefore the insertions and elimination of partial redundancy becomes incorrect. Our algorithm E-path_PRE remedies this correctness problem and computes anticipatability and availability correctly.

Other advantages compared to the SSA based approach of [C$^+$97] are as follows: E-path_PRE does not perform (even conceptually) hoisting followed by sinking of expressions in order to find placements which provide life-time optimality. Since it follows the approach of edge-placement [Dha88a, DP93], it uses edge-splitting in a demand-driven manner rather than as an a priori step in a pre-pass of optimization. This is beneficial on many counts: The effort of identifying critical edges can be avoided; the algorithm identifies critical edges which need to be split during the process of code insertion. This approach also avoids the drawback of [C$^+$97] wherein a computation may be inserted along all out-edges of a node, if all are critical edges, instead of being inserted at the end of the node. Correctness and minimality of the insertions performed by our approach follow from [DD95], where formal proofs of these properties are offered.

All steps in our algorithm are linear with respect to the number of basic blocks and edges in the PFG. The complexity of the algorithm is $O(n+e)$ for a program size of one. For a program size of m the complexity is $O(m(n+e))$, where n and e are the number of nodes and edges in G.

The E-path_SRLE algorithm is linear in terms of time complexity of optimization and simple to understand. Since it is an extension of the E-path_PRE, complex dataflow properties are avoided. The solution is straight forward. Although the algorithm for SRLE by Dhaneshwar and Dhamdhere [DD95] is a correct and nice concept itself, the algorithm is of quadratic complexity. As far as we know, these solutions for PRE and SRLE are the first linear-time correct algorithms.

6.2 Concluding Remarks

E-path_PRE and E-path_SRLE are the formulations of PRE and SRLE in SSA. As far as we know, E-path_PRE and E-path_SRLE are the first correct formulations of PRE and SRLE in SSA. In addition these are the first correct linear-complexity algorithms for PRE and SRLE. The optimal number of insertion of computations reduces the code size in comparison with SSAPRE. The simplicity and linearity property of our algorithms make their implementations easier than other techniques. We have implemented E-path_PRE and have tested it (see Appendix A). Implementation of conventional bit-vector techniques and

SSPARE of PRE are expensive and complex due to the data flows, which they compute. As a part of PRE and SRLE, we have developed algorithms for computing availability and anticipatability in SSA. As far as we know, our algorithms for computing availability and anticipatability for all expressions and for all the basicblocks in the program are the first SSA based algorithms and first linear-cost algorithm in the literature.

Appendix A

Implementation

We have developed a prototype for our algorithms of E-path_PRE and have tested it. We have implemented it in 'C' and on HP-UX B.11.00 U 9000/802 UNIX server. Below we have discussed about the interface, the salient features of our prototype and some results.

A.1 Interface

The input to our prototype is the program flow graph on which PRE has to be performed and the dominator tree information of the program flow graph. The prototype does not do any control flow analysis to generate PFG or dominator tree, if the source code of the program is given as input. Therefore it takes the program flow graph specification and the dom-tree information as the input.

The input files are passed as command line parameters to the program. If the executable is named **E-path_PRE**, then the command

E-path_PRE pfg.dat dom.dat

can be used to PRE-optimize the program, whose program flow graph and dominator tree information are given in pfg.dat and dom.dat respectively. A sample PFG is in figure 4.1, whose file specification to be given as input is given in figure A.1. The dom-tree for this example is given in figure A.2.

The specification syntax of the PFG input file (*pfg.dat*) is given as under.

- Each basicblock is demarcated by a keyword **basicblock**. It is followed by the basicblock number (see figure A.1).

```
basicblock 0 ;
predlist ;
succlist 1 2 ;
a 1 ;
b 1 ;

basicblock 1 ;
predlist 0 ;
succlist 8 ;
a 6 ;
t a * b ;

basicblock 2 loopentry ;
predlist 0 6 ;
succlist 3 4 5 ;
c 1 ;

basicblock 3 ;
predlist 2  ;
succlist 6 ;
c 2 ;

basicblock 4 ;
predlist 2  ;
succlist 6 ;
t a * b ;
a 2 ;
b 3 ;

basicblock 5 ;
predlist 2  ;
succlist 6 ;
t a * b ;

basicblock 6 ;
predlist 3 4 5 ;
succlist 7 2 ;
c 5 ;

basicblock 7 ;
predlist 6 ;
succlist 8 ;
t a * b ;
a 5 ;

basicblock 8 ;
predlist 1 7  ;
succlist  ;
c 4 ;
```

Figure A.1: Sample input file for program on program flow graph of figure 4.1 to our E-path_PRE prototype.

```
1 0
2 0
8 0
3 2
7 6
4 2
5 2
6 2
```

Figure A.2: Sample input file for dominator tree of the program flow graph of figure 4.1 to our E-path_PRE prototype.

- The basicblock number starts from 0 onwards. It is assumed that basicblock 0 is the entry of the program flow graph. The basicblock with the largest number is assumed as the exit of the program. In addition the basicblock numbers must be sequential between the 0 and the largest basicblock number.

- The basicblock specifier is followed by **predlist** keyword. This keyword is used to indicate the next list of numbers separated by spaces, are the list of predecessors of the current basicblock. In the figure A.1, for basicblock 0, there is no predecessor and therefore the list after **predlist** is followed by no number. For basicblock 6, the predecessors are basicblocks 3, 4 and 5. Each number is separated by a space.

- For successor list the keyword is **succlist**. The list of numbers must be followed by a semicolon **;**.

- After succlist, statements start. Each basicblock must have at least one statement. The statements are simple assignment statements. No function call nor statements with unary operator(s) is allowed. The left operand of the statement is written first and the assignment operator $=$ is implicit between it and the expression. For example, in figure A.1, in basicblock 4, the statement is $t = a * b$. This is mentioned as $t\ \ a * b$;. Note the space between the operand and the operator. No operand and operator must be adjacent without any space in between them.

- Each field is separated by a space. The **;** at the end of succlist line and statement lines indicates that the next line is another statement.

- The purpose of the **;** is to indicate that the next line is a statement. Therefore the lines of **basicblock** and **predlist** may not end with a **;**. All statements must end with a ;. But to conform to a more regular specification format, we have all the lines ending with a ; .

- If a basicblock is a loop entry, it must be specified after the basicblock number. In the figure A.1, the basicblock 2 is a loop entry. Note the way of specification.

For dominator tree data, we are describing the format below. The file format is :

- The format of each line is as : basicblock-number followed by the basicblock-number of immediate dominator. In the figure A.2, the basicblock 3 has the immediate dominator as basicblock 2 (see figure 4.2).
- All the numbers must be non-negative, otherwise the program may behave unpredictably.
- The entry basicblock (with number 0) must not have any entry here indicating its immediate dominator. This is because entry node is the dominator itself and it does not have any other dominator.

Some more constraints on the program and the program flow graph follow.

- It is presumed that all variables used/encountered in the program has at least one definition/assignment. This is to simplify the implementation and to avoid to detect such expressions and add dummy assignments at the entry of the PFG.
- The entry node also must not be a part of any loop. It is strongly recommended that first instruction of a program not to be a loop instruction.
- The convention that is a must is : the basicblock number has to start with 0 and the exit basicblock will have the last basicblockno. The number will be sequential from 0 to the last one.
- Every node must contain some statement.

In the result (or the output of the program), the versions given to a variable is shown in parenthesis on the right hand side of the variable. The $\perp$ version is shown as $-$ in the output SSA form.

A.2 Salient Features of E-path_PRE Prototype

The prototype takes the inputs and builds the program flow graph as well as the dominator tree. Then it generates the standard SSA form of the program (in which only program variables are given versions; no expression variable is introduced in it) and parallely also it generates the PRESSA form. We have merged the Φ-Insertion stage of PRESSA with ϕ-Insertion and Renaming of PRESSA with the renaming of standard SSA. Thereby we save two passes on the program. During renaming, the def-use chains are built up. The technique of specifically building the def-use chains during renaming is very simple : with each version of program variable or expression variable, associate the instruction that is its definition. Whenever a new version is given to a variable due to an instruction (which is the definition), the instruction (or rather its address in the memory) is pushed onto the corresponding renaming stack along with the version given. Next whenever the version is used later in the program, the definition of that variable is made the defining instruction. Next the availability and anticipatability are computed. To solve the correctness problem of SSAPRE (see section 2.2.2), we do not push the instruction as a whole to the renaming stack as we encounter them. But in another stack of *recenttype*, we hold the type[1]. This is done to give the same effect of the pushing the instructions onto the renaming stack as well as to reduce the memory usage by the stacks and to reduce the cost of push and pop. After insertion of computations, the def-use chains are updated appropriately.

A.3 Results

The following input has used the variable t and therefore the prototype does not use t as the temporary variable during elimination of redundancy. It uses w for this purpose. The internal representation of expressions such as $a * b$ is $a\$ * \$b$.

Test - 1

Input Program —

```
basicblock 0 ;
predlist ;
succlist 1 2 ;
a 1 ;
b 2 ;
t a * b ;

basicblock 1 loopentry ;
predlist 0 11 ;
succlist 3 4 ;
```

[1]Type of an instruction can be original occurrence, ϕ-function, Φ-function or inserted occurrence

```
a 3 ;

basicblock 2 ;
predlist  0 ;
succlist 5 6 ;
t 4 ;

basicblock 3 loopentry ;
predlist 10 1 ;
succlist 7 8 ;
t 3 ;

basicblock 4 ;
predlist  1 ;
succlist 11 ;
t a * b ;

basicblock 5 ;
predlist  2 ;
succlist 9 ;
t a * b ;

basicblock 6 ;
predlist  2 ;
succlist 9 ;
t 6 ;

basicblock 7 ;
predlist  3 ;
succlist 10 ;
t a * b ;

basicblock 8 ;
predlist  3 ;
succlist 10 ;
t 4 ;

basicblock 9 ;
predlist  5 6 ;
succlist 12 ;
t 4 ;

basicblock 10 ;
predlist  7 8 ;
succlist 3 11 ;
t 4 ;

basicblock 11 ;
predlist 10 4 ;
succlist 1 12 ;
t a * b ;

basicblock 12 ;
predlist   11 9 ;
succlist 13 14 ;
t 4 ;

basicblock 13 ;
predlist   12  ;
succlist 15 ;
```

```
t 4 ;

basicblock 14 ;
predlist   12  ;
succlist 15 ;
t 4 ;

basicblock 15 ;
predlist   13 14  ;
succlist  ;
t 4 ;
```

Input Dominator tree —

```
1 0
2 0
12 0
3 1
4 1
11 1
7 3
10 3
8 3
5 2
9 2
6 2
13 12
14 12
15 12
```

Result —

```
--------------- Start : Dominance Frontier Information ---------------
Set of Dominance Frontier for basicblock-7 :  10
Set of Dominance Frontier for basicblock-10 :  3  11
Set of Dominance Frontier for basicblock-8 :  10
Set of Dominance Frontier for basicblock-3 :  3  11
Set of Dominance Frontier for basicblock-4 :  11
Set of Dominance Frontier for basicblock-11 :  1  12
Set of Dominance Frontier for basicblock-1 :  1  12
Set of Dominance Frontier for basicblock-5 :  9
Set of Dominance Frontier for basicblock-9 :  12
Set of Dominance Frontier for basicblock-6 :  9
Set of Dominance Frontier for basicblock-2 :  12
Set of Dominance Frontier for basicblock-13 :  15
Set of Dominance Frontier for basicblock-14 :  15
Set of Dominance Frontier for basicblock-15 :
Set of Dominance Frontier for basicblock-12 :
Set of Dominance Frontier for basicblock-0 :
---------------- End : Dominance Frontier Information ----------------
------------------ Start PFG : after Phi-Insertion -------------------
No. of basicblocks = 16, No. of edges = 22, No. of instructions = 18
---------------------------------------------------------------------

BasicBlock : 0, No. of instructions = 3
-----------------------------------------
stmtno-1 a = 1;
stmtno-2 b = 2;
```

```
stmtno-3 t = a * b;

BasicBlock : 1, No. of instructions = 4
-----------------------------------------
stmtno-4 t =  phi(t , t);
stmtno-3 a =  phi(a , a);
stmtno-2 h =  Phi(h , h);
stmtno-1 a = 3;

BasicBlock : 2, No. of instructions = 1
-----------------------------------------
stmtno-1 t = 4;

BasicBlock : 3, No. of instructions = 3
-----------------------------------------
stmtno-2 t =  phi(t , t);
stmtno-3 h =  Phi(h , h);
stmtno-1 t = 3;

BasicBlock : 4, No. of instructions = 1
-----------------------------------------
stmtno-1 t = a * b;

BasicBlock : 5, No. of instructions = 1
-----------------------------------------
stmtno-1 t = a * b;

BasicBlock : 6, No. of instructions = 1
-----------------------------------------
stmtno-1 t = 6;

BasicBlock : 7, No. of instructions = 1
-----------------------------------------
stmtno-1 t = a * b;

BasicBlock : 8, No. of instructions = 1
-----------------------------------------
stmtno-1 t = 4;

BasicBlock : 9, No. of instructions = 3
-----------------------------------------
stmtno-2 t =  phi(t , t);
stmtno-3 h =  Phi(h , h);
stmtno-1 t = 4;

BasicBlock : 10, No. of instructions = 3
-----------------------------------------
stmtno-2 t =  phi(t , t);
stmtno-3 h =  Phi(h , h);
stmtno-1 t = 4;

BasicBlock : 11, No. of instructions = 3
-----------------------------------------
stmtno-2 t =  phi(t , t);
stmtno-3 h =  Phi(h , h);
stmtno-1 t = a * b;

BasicBlock : 12, No. of instructions = 4
-----------------------------------------
stmtno-4 t =  phi(t , t);
```

```
stmtno-3 a =  phi(a , a);
stmtno-2 h =  Phi(h , h);
stmtno-1 t = 4;

BasicBlock : 13, No. of instructions = 1
-----------------------------------------
stmtno-1 t = 4;

BasicBlock : 14, No. of instructions = 1
-----------------------------------------
stmtno-1 t = 4;

BasicBlock : 15, No. of instructions = 2
-----------------------------------------
stmtno-2 t =  phi(t , t);
------------------ End PFG : after Phi-Insertion ---------------------

--------------------- Start PFG : PRESSA form ------------------------
No. of basicblocks = 16, No. of edges = 22, No. of instructions = 18
----------------------------------------------------------------------

BasicBlock : 0, No. of instructions = 6
-----------------------------------------
stmtno-6 t(1) = 0;
stmtno-5 b(1) = 0;
stmtno-4 a(1) = 0;
stmtno-1 a(2) = 1;
stmtno-2 b(2) = 2;
stmtno-3 t(2) = a(2) * b(2); [h(1)]

BasicBlock : 1, No. of instructions = 4
-----------------------------------------
stmtno-4 t(3) =  phi(t(2) , t(12));
stmtno-3 a(3) =  phi(a(2) , a(4));
stmtno-2 h(2) =  Phi(h(1) , h(6));
stmtno-1 a(4) = 3;

BasicBlock : 2, No. of instructions = 1
-----------------------------------------
stmtno-1 t(13) = 4;

BasicBlock : 3, No. of instructions = 3
-----------------------------------------
stmtno-2 t(4) =  phi(t(8) , t(3));
stmtno-3 h(3) =  Phi(h(4) , -);
stmtno-1 t(5) = 3;

BasicBlock : 4, No. of instructions = 1
-----------------------------------------
stmtno-1 t(10) = a(4) * b(2); [h(5)]

BasicBlock : 5, No. of instructions = 1
-----------------------------------------
stmtno-1 t(14) = a(2) * b(2); [h(1)]

BasicBlock : 6, No. of instructions = 1
-----------------------------------------
stmtno-1 t(17) = 6;

BasicBlock : 7, No. of instructions = 1
```

```
-----------------------------------------
stmtno-1 t(6) = a(4) * b(2); [h(3)]

BasicBlock : 8, No. of instructions = 1
-----------------------------------------
stmtno-1 t(9) = 4;

BasicBlock : 9, No. of instructions = 3
-----------------------------------------
stmtno-2 t(15) =  phi(t(14) , t(17));
stmtno-3 h(7) =  Phi(h(1) , h(1));
stmtno-1 t(16) = 4;

BasicBlock : 10, No. of instructions = 3
-----------------------------------------
stmtno-2 t(7) =  phi(t(6) , t(9));
stmtno-3 h(4) =  Phi(h(3) , h(3));
stmtno-1 t(8) = 4;

BasicBlock : 11, No. of instructions = 3
-----------------------------------------
stmtno-2 t(11) =  phi(t(8) , t(10));
stmtno-3 h(6) =  Phi(h(4) , h(5));
stmtno-1 t(12) = a(4) * b(2); [h(6)]

BasicBlock : 12, No. of instructions = 4
-----------------------------------------
stmtno-4 t(18) =  phi(t(12) , t(16));
stmtno-3 a(5) =  phi(a(4) , a(2));
stmtno-2 h(8) =  Phi(h(6) , h(7));
stmtno-1 t(19) = 4;

BasicBlock : 13, No. of instructions = 1
-----------------------------------------
stmtno-1 t(20) = 4;

BasicBlock : 14, No. of instructions = 1
-----------------------------------------
stmtno-1 t(21) = 4;

BasicBlock : 15, No. of instructions = 2
-----------------------------------------
stmtno-2 t(22) =  phi(t(20) , t(21));
---------------------- End PFG : PRESSA form -------------------------

---------------------- Start : Node_type  ---------------------------

Basicblock-0 :
Expression - a$*$b : node_type - AVAIL
-----------------------------------------

Basicblock-1 :
Expression - a$*$b : node_type - OTHERS
-----------------------------------------

Basicblock-2 :
Expression - a$*$b : node_type - EMPTY
-----------------------------------------

Basicblock-3 :
```

```
Expression - a$*$b : node_type - EMPTY
-----------------------------------------

Basicblock-4 :
Expression - a$*$b : node_type - BOTH
-----------------------------------------

Basicblock-5 :
Expression - a$*$b : node_type - BOTH
-----------------------------------------

Basicblock-6 :
Expression - a$*$b : node_type - EMPTY
-----------------------------------------

Basicblock-7 :
Expression - a$*$b : node_type - BOTH
-----------------------------------------

Basicblock-8 :
Expression - a$*$b : node_type - EMPTY
-----------------------------------------

Basicblock-9 :
Expression - a$*$b : node_type - EMPTY
-----------------------------------------

Basicblock-10 :
Expression - a$*$b : node_type - EMPTY
-----------------------------------------

Basicblock-11 :
Expression - a$*$b : node_type - BOTH
-----------------------------------------

Basicblock-12 :
Expression - a$*$b : node_type - EMPTY
-----------------------------------------

Basicblock-13 :
Expression - a$*$b : node_type - EMPTY
-----------------------------------------

Basicblock-14 :
Expression - a$*$b : node_type - EMPTY
-----------------------------------------

Basicblock-15 :
Expression - a$*$b : node_type - EMPTY
-----------------------------------------
----------------------- End : Node_type  ----------------------------

-------------------- Start : has_real_use values ---------------------
Basicblock-0, Expression-a$*$b :
.............................................
Basicblock-1, Expression-a$*$b :
predecessor-0, Along this pred. has_real_use - 1
predecessor-11, Along this pred. has_real_use - 1
.............................................
Basicblock-2, Expression-a$*$b :
```

```
predecessor-0, Along this pred. has_real_use - 0
............................................
Basicblock-3, Expression-a$*$b :
predecessor-10, Along this pred. has_real_use - 0
predecessor-1, Along this pred. has_real_use - 0
............................................
Basicblock-4, Expression-a$*$b :
predecessor-1, Along this pred. has_real_use - 0
............................................
Basicblock-5, Expression-a$*$b :
predecessor-2, Along this pred. has_real_use - 0
............................................
Basicblock-6, Expression-a$*$b :
predecessor-2, Along this pred. has_real_use - 0
............................................
Basicblock-7, Expression-a$*$b :
predecessor-3, Along this pred. has_real_use - 0
............................................
Basicblock-8, Expression-a$*$b :
predecessor-3, Along this pred. has_real_use - 0
............................................
Basicblock-9, Expression-a$*$b :
predecessor-5, Along this pred. has_real_use - 1
predecessor-6, Along this pred. has_real_use - 1
............................................
Basicblock-10, Expression-a$*$b :
predecessor-7, Along this pred. has_real_use - 1
predecessor-8, Along this pred. has_real_use - 0
............................................
Basicblock-11, Expression-a$*$b :
predecessor-10, Along this pred. has_real_use - 0
predecessor-4, Along this pred. has_real_use - 1
............................................
Basicblock-12, Expression-a$*$b :
predecessor-11, Along this pred. has_real_use - 1
predecessor-9, Along this pred. has_real_use - 0
............................................
Basicblock-13, Expression-a$*$b :
predecessor-12, Along this pred. has_real_use - 0
............................................
Basicblock-14, Expression-a$*$b :
predecessor-12, Along this pred. has_real_use - 0
............................................
Basicblock-15, Expression-a$*$b :
predecessor-13, Along this pred. has_real_use - 0
predecessor-14, Along this pred. has_real_use - 0
............................................
-------------------- End : has_real_use values -----------------------

----------------- Start : Availability after Renaming ----------------

Basicblock - 0 :
Expression-a$*$b, avail-at-entry - 1

Basicblock - 1 :
Expression-a$*$b, avail-at-entry - 1

Basicblock - 2 :
Expression-a$*$b, avail-at-entry - 1
```

```
Basicblock - 3 :
Expression-a$*$b, avail-at-entry - 0

Basicblock - 4 :
Expression-a$*$b, avail-at-entry - 1

Basicblock - 5 :
Expression-a$*$b, avail-at-entry - 1

Basicblock - 6 :
Expression-a$*$b, avail-at-entry - 1

Basicblock - 7 :
Expression-a$*$b, avail-at-entry - 1

Basicblock - 8 :
Expression-a$*$b, avail-at-entry - 1

Basicblock - 9 :
Expression-a$*$b, avail-at-entry - 1

Basicblock - 10 :
Expression-a$*$b, avail-at-entry - 1

Basicblock - 11 :
Expression-a$*$b, avail-at-entry - 1

Basicblock - 12 :
Expression-a$*$b, avail-at-entry - 1

Basicblock - 13 :
Expression-a$*$b, avail-at-entry - 1

Basicblock - 14 :
Expression-a$*$b, avail-at-entry - 1

Basicblock - 15 :
Expression-a$*$b, avail-at-entry - 1

------------------ End : Availability after Renaming -----------------

------------ Start : Availability after Reset-availability -----------

Basicblock - 0 :
Expression-a$*$b, avail-at-entry - 1

Basicblock - 1 :
Expression-a$*$b, avail-at-entry - 1

Basicblock - 2 :
Expression-a$*$b, avail-at-entry - 1

Basicblock - 3 :
Expression-a$*$b, avail-at-entry - 0

Basicblock - 4 :
Expression-a$*$b, avail-at-entry - 1

Basicblock - 5 :
Expression-a$*$b, avail-at-entry - 1
```

```
Basicblock - 6 :
Expression-a$*$b, avail-at-entry - 1

Basicblock - 7 :
Expression-a$*$b, avail-at-entry - 1

Basicblock - 8 :
Expression-a$*$b, avail-at-entry - 1

Basicblock - 9 :
Expression-a$*$b, avail-at-entry - 1

Basicblock - 10 :
Expression-a$*$b, avail-at-entry - 0

Basicblock - 11 :
Expression-a$*$b, avail-at-entry - 0

Basicblock - 12 :
Expression-a$*$b, avail-at-entry - 1

Basicblock - 13 :
Expression-a$*$b, avail-at-entry - 1

Basicblock - 14 :
Expression-a$*$b, avail-at-entry - 1

Basicblock - 15 :
Expression-a$*$b, avail-at-entry - 1
------------ End : Availability after Reset-availability ------------

-------------------- Start : Final availability ----------------------

Basicblock - 0 :
Expression-a$*$b, avail-at-entry - 0, avail-at-exit - 1

Basicblock - 1 :
Expression-a$*$b, avail-at-entry - 1, avail-at-exit - 0

Basicblock - 2 :
Expression-a$*$b, avail-at-entry - 1, avail-at-exit - 1

Basicblock - 3 :
Expression-a$*$b, avail-at-entry - 0, avail-at-exit - 0

Basicblock - 4 :
Expression-a$*$b, avail-at-entry - 0, avail-at-exit - 1

Basicblock - 5 :
Expression-a$*$b, avail-at-entry - 1, avail-at-exit - 1

Basicblock - 6 :
Expression-a$*$b, avail-at-entry - 1, avail-at-exit - 1

Basicblock - 7 :
Expression-a$*$b, avail-at-entry - 0, avail-at-exit - 1

Basicblock - 8 :
Expression-a$*$b, avail-at-entry - 0, avail-at-exit - 0
```

```
Basicblock - 9 :
Expression-a$*$b, avail-at-entry - 1, avail-at-exit - 1

Basicblock - 10 :
Expression-a$*$b, avail-at-entry - 0, avail-at-exit - 0

Basicblock - 11 :
Expression-a$*$b, avail-at-entry - 0, avail-at-exit - 1

Basicblock - 12 :
Expression-a$*$b, avail-at-entry - 1, avail-at-exit - 1

Basicblock - 13 :
Expression-a$*$b, avail-at-entry - 1, avail-at-exit - 1

Basicblock - 14 :
Expression-a$*$b, avail-at-entry - 1, avail-at-exit - 1

Basicblock - 15 :
Expression-a$*$b, avail-at-entry - 1, avail-at-exit - 1
-------------------- End : Final availability ------------------------

------------------ Start : Before applying DownSafety ----------------

Basicblock - 0 :
Expression - a$*$b, ant-at-entry - 1

Basicblock - 1 :
Expression - a$*$b, ant-at-entry - 0

Basicblock - 2 :
Expression - a$*$b, ant-at-entry - 1

Basicblock - 3 :
Expression - a$*$b, ant-at-entry - 1

Basicblock - 4 :
Expression - a$*$b, ant-at-entry - 1

Basicblock - 5 :
Expression - a$*$b, ant-at-entry - 1

Basicblock - 6 :
Expression - a$*$b, ant-at-entry - 1

Basicblock - 7 :
Expression - a$*$b, ant-at-entry - 1

Basicblock - 8 :
Expression - a$*$b, ant-at-entry - 1

Basicblock - 9 :
Expression - a$*$b, ant-at-entry - 1

Basicblock - 10 :
Expression - a$*$b, ant-at-entry - 1

Basicblock - 11 :
Expression - a$*$b, ant-at-entry - 1
```

```
Basicblock - 12 :
Expression - a$*$b, ant-at-entry - 0

Basicblock - 13 :
Expression - a$*$b, ant-at-entry - 1

Basicblock - 14 :
Expression - a$*$b, ant-at-entry - 1

Basicblock - 15 :
Expression - a$*$b, ant-at-entry - 1
------------------- End : Before applying DownSafety -----------------

------------------ Start : After applying DownSafety -----------------

Basicblock - 0 :
Expression - a$*$b, ant-at-entry - 1

Basicblock - 1 :
Expression - a$*$b, ant-at-entry - 0

Basicblock - 2 :
Expression - a$*$b, ant-at-entry - 1

Basicblock - 3 :
Expression - a$*$b, ant-at-entry - 1

Basicblock - 4 :
Expression - a$*$b, ant-at-entry - 1

Basicblock - 5 :
Expression - a$*$b, ant-at-entry - 1

Basicblock - 6 :
Expression - a$*$b, ant-at-entry - 1

Basicblock - 7 :
Expression - a$*$b, ant-at-entry - 1

Basicblock - 8 :
Expression - a$*$b, ant-at-entry - 1

Basicblock - 9 :
Expression - a$*$b, ant-at-entry - 0

Basicblock - 10 :
Expression - a$*$b, ant-at-entry - 1

Basicblock - 11 :
Expression - a$*$b, ant-at-entry - 1

Basicblock - 12 :
Expression - a$*$b, ant-at-entry - 0

Basicblock - 13 :
Expression - a$*$b, ant-at-entry - 1

Basicblock - 14 :
Expression - a$*$b, ant-at-entry - 1
```

```
Basicblock - 15 :
Expression - a$*$b, ant-at-entry - 1
------------------- End : After applying DownSafety ------------------

------------------- Start : Final anticipatability -------------------

Basicblock - 0 :
Expression - a$*$b, ant-at-entry - 0, ant-at-exit - 0

Basicblock - 1 :
Expression - a$*$b, ant-at-entry - 0, ant-at-exit - 1

Basicblock - 2 :
Expression - a$*$b, ant-at-entry - 0, ant-at-exit - 0

Basicblock - 3 :
Expression - a$*$b, ant-at-entry - 1, ant-at-exit - 1

Basicblock - 4 :
Expression - a$*$b, ant-at-entry - 1, ant-at-exit - 1

Basicblock - 5 :
Expression - a$*$b, ant-at-entry - 1, ant-at-exit - 0

Basicblock - 6 :
Expression - a$*$b, ant-at-entry - 0, ant-at-exit - 0

Basicblock - 7 :
Expression - a$*$b, ant-at-entry - 1, ant-at-exit - 1

Basicblock - 8 :
Expression - a$*$b, ant-at-entry - 1, ant-at-exit - 1

Basicblock - 9 :
Expression - a$*$b, ant-at-entry - 0, ant-at-exit - 0

Basicblock - 10 :
Expression - a$*$b, ant-at-entry - 1, ant-at-exit - 1

Basicblock - 11 :
Expression - a$*$b, ant-at-entry - 1, ant-at-exit - 0

Basicblock - 12 :
Expression - a$*$b, ant-at-entry - 0, ant-at-exit - 0

Basicblock - 13 :
Expression - a$*$b, ant-at-entry - 0, ant-at-exit - 0

Basicblock - 14 :
Expression - a$*$b, ant-at-entry - 0, ant-at-exit - 0

Basicblock - 15 :
Expression - a$*$b, ant-at-entry - 0, ant-at-exit - 0

------------------- End : Final anticipatability ---------------------
----------------- PRE started for Expression : a$*$b -----------------

--------------- Start : Egraph with useless nodes if any -------------
No. of nodes in Egraph - 18
```

```
------------------------------

Basicblock 0 :OUT Node of basicblock 0 :
..............................
Predecessor (No. of predessors = 0) basicblocks :
Successor (No. of successors = 1) basicblocks :  2
----------------------------------------------------

**** Any node corresponding to basicblock-1 is not in Egraph. ****

Basicblock 2 :OUT Node of basicblock 2 :
..............................
Predecessor (No. of predessors = 1) basicblocks :  0
Successor (No. of successors = 2) basicblocks :  5  6
----------------------------------------------------

Basicblock 3 :OUT Node of basicblock 3 :
..............................
Predecessor (No. of predessors = 1) basicblocks :  10
Successor (No. of successors = 2) basicblocks :  7  8
----------------------------------------------------

Basicblock 4 :OUT Node of basicblock 4 :
..............................
Predecessor (No. of predessors = 0) basicblocks :
Successor (No. of successors = 1) basicblocks :  11
----------------------------------------------------

Basicblock 5 :.......Basicblock 5 has been splitted.......
IN Node of basicblock 5 :
..............................
predecessor (No. of predecessor = 1) basicblocks :  2
successor (No. of successor = 0) basicblocks :
OUT Node of basicblock 5 :
..............................
Predecessor (No. of predessors = 0) basicblocks :
Successor (No. of successors = 1) basicblocks :  9
----------------------------------------------------

Basicblock 6 :OUT Node of basicblock 6 :
..............................
Predecessor (No. of predessors = 1) basicblocks :  2
Successor (No. of successors = 1) basicblocks :  9
----------------------------------------------------

Basicblock 7 :.......Basicblock 7 has been splitted.......
IN Node of basicblock 7 :
..............................
predecessor (No. of predecessor = 1) basicblocks :  3
successor (No. of successor = 0) basicblocks :
OUT Node of basicblock 7 :
..............................
Predecessor (No. of predessors = 0) basicblocks :
Successor (No. of successors = 1) basicblocks :  10
----------------------------------------------------

Basicblock 8 :OUT Node of basicblock 8 :
..............................
Predecessor (No. of predessors = 1) basicblocks :  3
Successor (No. of successors = 1) basicblocks :  10
```

```
----------------------------------------------------

Basicblock 9 :OUT Node of basicblock 9 :
..............................
Predecessor (No. of predessors = 2) basicblocks :  5  6
Successor (No. of successors = 1) basicblocks :  12
----------------------------------------------------

Basicblock 10 :OUT Node of basicblock 10 :
..............................
Predecessor (No. of predessors = 2) basicblocks :  7  8
Successor (No. of successors = 2) basicblocks :  3  11
----------------------------------------------------

Basicblock 11 :.......Basicblock 11 has been splitted.......
IN Node of basicblock 11 :
..............................
predecessor (No. of predecessor = 2) basicblocks :  10  4
successor (No. of successor = 0) basicblocks :
OUT Node of basicblock 11 :
..............................
Predecessor (No. of predessors = 0) basicblocks :
Successor (No. of successors = 1) basicblocks :  12
----------------------------------------------------

Basicblock 12 :OUT Node of basicblock 12 :
..............................
Predecessor (No. of predessors = 2) basicblocks :  11  9
Successor (No. of successors = 2) basicblocks :  13  14
----------------------------------------------------

Basicblock 13 :OUT Node of basicblock 13 :
..............................
Predecessor (No. of predessors = 1) basicblocks :  12
Successor (No. of successors = 1) basicblocks :  15
----------------------------------------------------

Basicblock 14 :OUT Node of basicblock 14 :
..............................
Predecessor (No. of predessors = 1) basicblocks :  12
Successor (No. of successors = 1) basicblocks :  15
----------------------------------------------------

Basicblock 15 :OUT Node of basicblock 15 :
..............................
Predecessor (No. of predessors = 2) basicblocks :  13  14
Successor (No. of successors = 0) basicblocks :
----------------------------------------------------
---------------- End : Egraph with useless nodes if any --------------

8 useless nodes have been removed.

------------------ Start : Egraph with no useless node ---------------
No. of nodes in Egraph - 10
------------------------------

Basicblock 0 :OUT Node of basicblock 0 :
..............................
Predecessor (No. of predessors = 0) basicblocks :
Successor (No. of successors = 1) basicblocks :  2
```

```
----------------------------------------------------

**** Any node corresponding to basicblock-1 is not in Egraph. ****

Basicblock 2 :OUT Node of basicblock 2 :
..............................
Predecessor (No. of predessors = 1) basicblocks :  0
Successor (No. of successors = 1) basicblocks :  5
----------------------------------------------------

Basicblock 3 :OUT Node of basicblock 3 :
..............................
Predecessor (No. of predessors = 1) basicblocks :  10
Successor (No. of successors = 2) basicblocks :  7  8
----------------------------------------------------

Basicblock 4 :OUT Node of basicblock 4 :
..............................
Predecessor (No. of predessors = 0) basicblocks :
Successor (No. of successors = 1) basicblocks :  11
----------------------------------------------------

Basicblock 5 :IN Node of basicblock 5 :
..............................
Predecessor (No. of predessors = 1) basicblocks :  2
Successor (No. of successors = 0) basicblocks :
----------------------------------------------------

**** Any node corresponding to basicblock-6 is not in Egraph. ****

Basicblock 7 :.......Basicblock 7 has been splitted.......
IN Node of basicblock 7 :
..............................
predecessor (No. of predecessor = 1) basicblocks :  3
successor (No. of successor = 0) basicblocks :
OUT Node of basicblock 7 :
..............................
Predecessor (No. of predessors = 0) basicblocks :
Successor (No. of successors = 1) basicblocks :  10
----------------------------------------------------

Basicblock 8 :OUT Node of basicblock 8 :
..............................
Predecessor (No. of predessors = 1) basicblocks :  3
Successor (No. of successors = 1) basicblocks :  10
----------------------------------------------------

**** Any node corresponding to basicblock-9 is not in Egraph. ****

Basicblock 10 :OUT Node of basicblock 10 :
..............................
Predecessor (No. of predessors = 2) basicblocks :  7  8
Successor (No. of successors = 2) basicblocks :  3  11
----------------------------------------------------

Basicblock 11 :IN Node of basicblock 11 :
..............................
Predecessor (No. of predessors = 2) basicblocks :  10  4
Successor (No. of successors = 0) basicblocks :
----------------------------------------------------
```

```
**** Any node corresponding to basicblock-12 is not in Egraph. ****

**** Any node corresponding to basicblock-13 is not in Egraph. ****

**** Any node corresponding to basicblock-14 is not in Egraph. ****

**** Any node corresponding to basicblock-15 is not in Egraph. ****
------------------ End : Egraph with no useless node -----------------

-------------------- Start : Insertion Points ------------------------
0'th insertion point along edge - (1,3)

--------------------- End : Insertion Points -------------------------

No uses ExtraneousPhi is in basicblock - 1
No uses ExtraneousPhi is in basicblock - 12

------------------- Start PFG : after Insertion ----------------------

No. of basicblocks = 17, No. of edges = 23, No. of instructions = 19
----------------------------------------------------------------------

BasicBlock : 0, No. of instructions = 6
------------------------------------------
stmtno-6 t(1) = 0;
stmtno-5 b(1) = 0;
stmtno-4 a(1) = 0;
stmtno-1 a(2) = 1;
stmtno-2 b(2) = 2;
stmtno-3 t(2) = a(2) * b(2); [h(1)]

BasicBlock : 1, No. of instructions = 4
------------------------------------------
stmtno-4 t(3) =  phi(t(2) , t(12));
stmtno-3 a(3) =  phi(a(2) , a(4));
stmtno-2 h(2) =  Phi(h(1) , h(6));
stmtno-1 a(4) = 3;

BasicBlock : 2, No. of instructions = 1
------------------------------------------
stmtno-1 t(13) = 4;

BasicBlock : 3, No. of instructions = 3
------------------------------------------
stmtno-2 t(4) =  phi(t(8) , t(3));
stmtno-3 h(3) =  Phi(h(4) , h(10));
stmtno-1 t(5) = 3;

BasicBlock : 4, No. of instructions = 1
------------------------------------------
stmtno-1 t(10) = a(4) * b(2); [h(5)]

BasicBlock : 5, No. of instructions = 1
------------------------------------------
stmtno-1 t(14) = a(2) * b(2); [h(1)]

BasicBlock : 6, No. of instructions = 1
------------------------------------------
stmtno-1 t(17) = 6;
```

```
BasicBlock : 7, No. of instructions = 1
-----------------------------------------
stmtno-1 t(6) = a(4) * b(2); [h(3)]

BasicBlock : 8, No. of instructions = 1
-----------------------------------------
stmtno-1 t(9) = 4;

BasicBlock : 9, No. of instructions = 3
-----------------------------------------
stmtno-2 t(15) =  phi(t(14) , t(17));
stmtno-3 h(7) =  Phi(h(1) , h(1));
stmtno-1 t(16) = 4;

BasicBlock : 10, No. of instructions = 3
-----------------------------------------
stmtno-2 t(7) =  phi(t(6) , t(9));
stmtno-3 h(4) =  Phi(h(3) , h(3));
stmtno-1 t(8) = 4;

BasicBlock : 11, No. of instructions = 3
-----------------------------------------
stmtno-2 t(11) =  phi(t(8) , t(10));
stmtno-3 h(6) =  Phi(h(4) , h(5));
stmtno-1 t(12) = a(4) * b(2); [h(6)]

BasicBlock : 12, No. of instructions = 4
-----------------------------------------
stmtno-4 t(18) =  phi(t(12) , t(16));
stmtno-3 a(5) =  phi(a(4) , a(2));
stmtno-2 h(8) =  Phi(h(6) , h(7));
stmtno-1 t(19) = 4;

BasicBlock : 13, No. of instructions = 1
-----------------------------------------
stmtno-1 t(20) = 4;

BasicBlock : 14, No. of instructions = 1
-----------------------------------------
stmtno-1 t(21) = 4;

BasicBlock : 15, No. of instructions = 2
-----------------------------------------
stmtno-2 t(22) =  phi(t(20) , t(21));

BasicBlock : 16, No. of instructions = 1
-----------------------------------------
stmtno-1 insvar(1) = a(4) * b(2); [h(10)]
------------------- End PFG : after Insertion ------------------------
------------ Start PFG : after elimination of redundancies -----------

No. of basicblocks = 17, No. of edges = 23, No. of instructions = 19
----------------------------------------------------------------------

BasicBlock : 0, No. of instructions = 6
-----------------------------------------
stmtno-6 t(1) = 0;
stmtno-5 b(1) = 0;
stmtno-4 a(1) = 0;
```

```
stmtno-1 a(2) = 1;
stmtno-2 b(2) = 2;
stmtno-3 t(2) = a(2) * b(2); [w(1)]

BasicBlock : 1, No. of instructions = 3
------------------------------------------
stmtno-4 t(3) =  phi(t(2) , t(12));
stmtno-3 a(3) =  phi(a(2) , a(4));
stmtno-1 a(4) = 3;

BasicBlock : 2, No. of instructions = 1
------------------------------------------
stmtno-1 t(13) = 4;

BasicBlock : 3, No. of instructions = 3
------------------------------------------
stmtno-2 t(4) =  phi(t(8) , t(3));
stmtno-3 w(3) =  phi(w(4) , w(10));
stmtno-1 t(5) = 3;

BasicBlock : 4, No. of instructions = 1
------------------------------------------
stmtno-1 t(10) = a(4) * b(2); [w(5)]

BasicBlock : 5, No. of instructions = 1
------------------------------------------
stmtno-1 t(14) = w(1); [w(1)]

BasicBlock : 6, No. of instructions = 1
------------------------------------------
stmtno-1 t(17) = 6;

BasicBlock : 7, No. of instructions = 1
------------------------------------------
stmtno-1 t(6) = w(3); [w(3)]

BasicBlock : 8, No. of instructions = 1
------------------------------------------
stmtno-1 t(9) = 4;

BasicBlock : 9, No. of instructions = 3
------------------------------------------
stmtno-2 t(15) =  phi(t(14) , t(17));
stmtno-3 w(7) =  phi(w(1) , w(1));
stmtno-1 t(16) = 4;

BasicBlock : 10, No. of instructions = 3
------------------------------------------
stmtno-2 t(7) =  phi(t(6) , t(9));
stmtno-3 w(4) =  phi(w(3) , w(3));
stmtno-1 t(8) = 4;

BasicBlock : 11, No. of instructions = 3
------------------------------------------
stmtno-2 t(11) =  phi(t(8) , t(10));
stmtno-3 w(6) =  phi(w(4) , w(5));
stmtno-1 t(12) = w(6); [w(6)]

BasicBlock : 12, No. of instructions = 3
------------------------------------------
```

```
stmtno-4 t(18) =  phi(t(12) , t(16));
stmtno-3 a(5) =  phi(a(4) , a(2));
stmtno-1 t(19) = 4;

BasicBlock : 13, No. of instructions = 1
------------------------------------------
stmtno-1 t(20) = 4;

BasicBlock : 14, No. of instructions = 1
------------------------------------------
stmtno-1 t(21) = 4;

BasicBlock : 15, No. of instructions = 2
------------------------------------------
stmtno-2 t(22) =  phi(t(20) , t(21));

BasicBlock : 16, No. of instructions = 1
------------------------------------------
stmtno-1 insvar(1) = a(4) * b(2); [w(10)]
------------ End PFG : after elimination of redundancies -------------
PRE finished for Expression : a$*$b
No more Expressions.... Above is the final PFG...
---------------------- End Final PFG : after PRE ---------------------
```

References

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers—Principles, Techniques and Tools.* Addison Wesley, 1986.

[BCS] Preston Briggs, Keith D. Cooper, and L. Taylor Simpson. Practical Improvements to the Construction and Destruction of Static Single Assignment Form. *To appear in Software Practice and Experience.*

[BM] Marc M Brandis and H Mössenböck. Single-Pass Generation of Static Single-Assignment Form for Structured Languages. *ACM Transations on Programming Languages and Systems*, 16(6).

[C$^+$91] R. Cytron et al. Efficiently computing static single assignment form and the control dependence graph. *ACM Transaction on Programming Languages and Systems*, 13(4):451–490, 1991.

[C$^+$97] F. Chow et al. A new algorithm for partial redundancy elimination based on SSA form. *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 273–286, June 1997.

[DD95] V. M. Dhaneshwar and D. M. Dhamdhere. Strength reduction of large expressions. *Journal of Programming Languages*, 3:95–120, 1995.

[Dha88a] D. M. Dhamdhere. A fast algorithm for code movement optimization. *SIGPLAN Notices*, 23(10):172–180, 1988.

[Dha88b] D. M. Dhamdhere. Register assignment using code placement techniques. *Computer Languages*, 13(2):75–93, 1988.

[Dha89] D. M. Dhamdhere. A new algorithm for composite hoisting and strength reduction optimisation. *International Journal of Computer Mathematics*, 27:1–14, 1989.

[Dha90a] D. M. Dhamdhere. *Compiler Construction, Principles and Practices.* MacMillan India Limited, 1990.

[Dha90b] D. M. Dhamdhere. A usually linear algorithm for register assignment using edge placement of load and store instructions. *Computer Languages*, 15(2):83–94, 1990.

[DK93] D. M. Dhamdhere and U. P. Khedker. Complexity of bidirectional data flows. *Proceedings of Twentieth annual symposium on the Principles of Programming Languages*, pages 397–408, 1993.

[DP93] D. M. Dhamdhere and H. Patil. An elimination algorithm for bi-directional data flow analysis using edge placement technique. *ACM Transactions on Programming Languages and Systems*, 15(2):312–336, 1993.

[DRZ92] D. M. Dhamdhere, B. K. Rosen, and F. K. Zadeck. How to analyze large programs efficiently and informatively. *ACM SIGPLAN '92 Conference on Programming Languages Design and Implementation*, pages 212–223, 1992.

[DS88] K. Drechsler and M. P. Stadel. A solution to a problem with morel and renvoise's "global optimization by suppression of partial redundancies". *ACM Transactions on Programming Languages and Systems*, 10(4):635–640, 1988.

[DS93] K. Drechsler and M. P. Stadel. A variation of Knoop, Ruthing, and Steffen's lazy code motion. *SIGPLAN Notices*, 28(5):29–38, 1993.

[JD82] S. M. Joshi and D. M. Dhamdhere. A composite hoisting-strength reduction transformation for global program optimisation, Parts I and II. *International Journal of Computer Mathematics*, 11(1 & 2):21–24 & 111–126, 1982.

[JPP94] R. Johnson, D. Pearson, and K. Pingali. The program structure tree: Computing control regions in linear time. *Proceedings of ACM SIGPLAN '94 Conference on Programming Languages Design and Implementation*, pages 171–185, June 1994.

[K$^+$98] R. Kennedy et al. Strength Reduction via SSAPRE. *Lecture notes in computer science*, 1383:144–157, 1998.

[KD] Ashish Kundu and D. M. Dhamdhere. E-path_PRE: SSA based partial redundancy elimination using eliminatability paths. *Submitted to ACM SIGPLAN 2000 Conference on Programming Languages Design and Implementation.*

[KD94] U. P. Khedker and D. M. Dhamdhere. A generalised theory of bit vector data flow analysis. *ACM Transactions on Programming Languages and Systems*, 16(5):1472–1511, September 1994.

[KD99] U. P. Khedker and D. M. Dhamdhere. Bidirectional data flow analysis : Myths and reality. *SIGPLAN Notices*, 34(6):47–57, 1999.

[KRS92] J. Knoop, O. Ruthing, and B. Steffen. Lazy code motion. *ACM SIGPLAN '92 Conference on Programming Languages Design and Implementation*, pages 224–234, June 1992.

[L$^+$98] R. Lo et al. Register promotion by sparse partial redundancy elimination of loads and stores. *ACM SIGPLAN '98 Conference on Programming Languages Design and Implementation*, pages 26–37, May 1998.

[MR79] E. Morel and C. Renvoise. Global optimization by suppression of partial rendundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[Muc97] S. S. Muchnick. *Advanced Compiler Design & Implementation.* Morgan Kaufmann, 1997.

[SG88] V. Sreedhar and G. Gao. A linear time algorithm for placing ϕ-nodes. *Conference Record of Eighteenth ACM Symposium on Principles of Programming Languages*, pages 12–27, January 1988.