E-path_PRE : SSA based partial redundancy elimination using eliminatability paths

Ashish Kundu and D. M. Dhamdhere*
ashishk@cse.iitb.ernet.in dmd@cse.iitb.ernet.in
Department of Computer Science and Engineering
Indian Institute of Technology, Bombay 400 076 (India).

1 Introduction

Partial redundancy elimination (PRE) is a powerful optimization technique which subsumes many classical optimizations like code movement, loop invariant movement and common subexpression elimination. The original formulation of PRE by Morel-Renvoise [22] involved complex bi-directional data flows and suffered from problems of redundant code movement and missed opportunities of optimization [4, 12]. 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 [4, 12, 10, 20, 8, 13, 18, 11, 19]. The basic PRE framework has also been extended to include strength reduction optimization [15, 6, 11, 17] and to use it for other applications like live range determination in register assignment [5, 7, 21].

Static single assignment (SSA) is a program representation used in modern optimizing compilers [3], whose appeal stems from the concise representation of def-use information. [10] had addressed the issue of using the SSA form for PRE optimizations. [2] have proposed a PRE algorithm based on the SSA form. They use the lazy code motion approach of [20] and cast it in an SSA-based algorithm for PRE. However, their algorithm is very complex and has a few deficiencies discussed later in Section 4.

This paper presents a formulation of SSA-based partial redundancy elimination which uses the PRE approach of [11]. A simplified overview of our approach is as follows: We identify eliminatability paths for an expression e in a program P. Let $[b_i \ \ldots \ b_k]$ be such a path. This path contains occurrences of e in nodes b_i and b_k which are downwards and upwards exposed [1, 23], respectively, and e is either available or anticipatable (i.e. very busy) at the exit of each node in the path $[b_i \ldots b_k)$. We eliminate the occurrence of e in node b_k by inserting computations of e in all nodes b_l such that some node along the path $(b_i \ \ldots \ b_k]$ is a successor of b_l.

Compared to earlier PRE work, and specifically the SSA-based approach of [2], the advantages of our approach are simplicity, understandability and efficiency. Our approach uses only well-known fundamental data flows of available expressions and anticipatable (i.e. very-busy) expressions [1, 23]. Unlike earlier approaches [4, 9, 10, 20], it does not perform (even conceptually) hoisting followed by sinking of expressions in order to find placements which provide life-time optimality. Hence it does not involve use of complex data flows or steps which affect its simplicity and understandability. It follows the approach of edge-placement [4, 9], which employs edge-splitting in a demand-driven manner rather than performing it in a pre-pass of optimization. Cumulatively, these aspects reduce the amount of work compared to the SSA-based approach of [2].

We describe fundamentals of the eliminatability path approach to PRE in Section 2. Section 3 presents details of our algorithm, which we call E-path_PRE algorithm, and proofs of the properties which form its basis. We illustrate the operation of the algorithm with the help of an example. Section 4 contains a comparison of our approach with other PRE approaches.

2 Partial redundancy elimination using eliminatability paths

We assume that a program is represented in the form of a program flow graph $G = (N, E, n_0)$, where N is the set of nodes (that is, basic blocks) in the

*Corresponding author

program, E is the set of control flow edges, and n_0 is the entry node of the program. The notation and definitions of the basic control and data flow concepts can be found in [1, 23]. This section summarizes the optimization approach based on the notion of eliminatability paths (e-paths). All material except the notion of e-paths is adapted from [11].

Terminology

An expression e is *locally available* in node b_i if b_i contains a *downwards exposed* occurrence of e, that is, an occurrence of e not followed by a definition of any of its operands. An expression e is *available* (*partially available*) at a program point if along all paths (along some path) from the start node to that point, there exists a computation of e not followed by a definition of any of its operands. A computation of e at program point w is *redundant* if e is available at w, and *partially redundant* if it is partially available at w.

An expression e is *locally anticipatable* in node b_i if b_i contains an *upwards exposed* occurrence of e, that is, an occurrence of e not preceded by a definition of any of its operands. e is *anticipatable* at a program point if each path starting at that point contains a computation of e not preceded by a definition of any of its operands. An expression e is *safe* at a point if it is either anticipatable or available at that point [16]. A generally accepted requirement of a hoisting algorithm is that it should place computations of an expression e only at points where e is safe.

Eliminatability paths (e-paths)

A node b_k is *empty* with respect to an expression e if b_k does not contain an occurrence of e, or definition(s) of any of its operands.

Definition 2.1 *An occurrence of an expression e in a node b_k is an* eliminatable occurrence of e *if the occurrence is locally anticipatable in b_k and there exists a path $[b_i \ldots b_k]$ such that :*

a. e is locally available at the exit of b_i,

b. All nodes on the path $(b_i \ldots b_k)$ are empty with respect to e, and

c. e is safe at the exit of each node on the path $[b_i \ldots b_k)$.

An expression e is *eliminatable in a node b_k* iff there is an eliminatable occurrence of e in node b_k or a computation of e placed at the exit of b_k would be an eliminatable occurrence of e. Note that eliminatability of e in node b_k requires the existence of some path $[b_i \ldots b_k]$ which satisfies Def. 2.1. Other paths reaching b_k need not satisfy Def. 2.1.

Definition 2.2 *A path $[b_i \ldots b_k]$ is an* eliminatability path (e-path) *for expression e if it satisfies conditions (a)–(c) of Def. 2.1.*

2.1 Elimination of partial redundancies

Eliminatable occurrences of an expression e can be eliminated after placing occurrences of e in the set of program nodes $\{b_l\}$ and the set of synthetic nodes $\{b_{l-m}\}$ as defined by DPH–1 and DPH–2 :

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

a. e is not available at the exit of b_l,

b. e is not eliminatable in b_l, and

c. All paths starting at the exit of b_l have a prefix $[b_l \ldots b_k]$ such that b_k contains an eliminatable occurrence of e, and for all nodes b_j on the path $(b_l \ldots b_k)$, 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

a. e cannot be placed in b_l due to the violation of condition DPH–1(c),

b. e is neither available nor eliminatable at the exit of node b_l, and

c. All paths starting at the entry of node b_m have a prefix $[b_m \ldots b_k]$ such that b_k contains an eliminatable occurrence of e, and for all nodes b_j on the path $[b_m \ldots b_k)$, e is eliminatable from b_j and b_j is empty with respect to e.

Early algorithms for hoisting and strength reduction suffer from the problem of *redundant hoisting*, that is hoisting not accompanied by execution profits (see [4] for an example). We can prove that the placement performed by DPH–1 and DPH–2 does not lead to redundant hoisting as follows: Consider an e-path $[b_i \ldots b_k]$ for expression e such that e is placed in a node b_l (original node or synthetic node) to eliminate the eliminatable occurrence of e in b_k. Let b_s be a successor of b_l along the path $[b_l \ldots b_k]$. From criteria (c) of DPH–1 and DPH–2 it follows that e is eliminatable from b_s. Hence

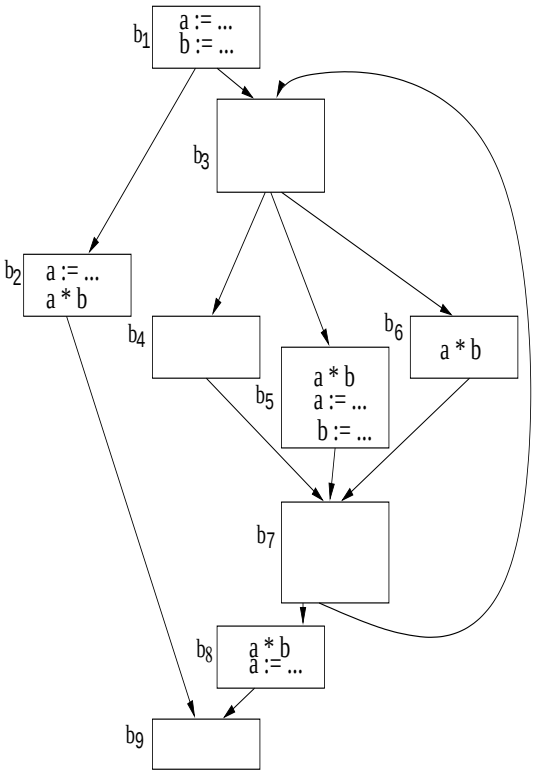

Figure 1: Optimization using e-paths

placing e in b_l instead of b_s does not constitute redundant hoisting.

Figure 1 contains an example of optimization using this approach. Computations in nodes b_5, b_6 and b_8 are eliminatable. Four e-paths exist in the program: b_6-b_7-b_3-b_6, b_6-b_7-b_3-b_5, b_6-b_7-b_3-b_4-b_7-b_8 and b_6-b_7-b_8. While optimizing the e-path b_6-b_7-b_3-b_6, placement is considered in nodes b_4, b_5 and along edge (b_1, b_3). Expression e is eliminatable in node b_4 due to existence of the e-path b_6-b_7-b_3-b_4-b_7-b_8, hence it is not placed there, however placement is needed in node b_5 and along edge (b_1, b_3). Consideration of path b_6-b_7-b_3-b_5 does not lead to any new insertions since node b_5 and edge (b_1, b_3) have already been identified as insertion points. Consideration of paths b_6-b_7-b_3-b_4-b_7-b_8 and b_6-b_7-b_8 similarly does not add any more insertion points. As a consequence, computations of $a * b$ are inserted in node b_5 and edge (b_1, b_3) and computations of $a * b$ in nodes b_5, b_6 and b_8 are eliminated.

3 The E-path_PRE algorithm

In the SSA form of a program, each variable has a single assignment and each use of the variable is dominated by this assignment. While converting a program to the SSA form, disjoint uses of a variable in the program are given different version numbers, which are written as subscripts of the variable. Wherever necessary different versions of a variable are merged by inserting ϕ functions to maintain the SSA property [3]. Uses of a variable in the original program are now renamed to use an appropriate version of the variable. Thus, occurrences of an expression $a * b$ may be replaced by $a_1 * b_1$ and $a_2 * b_1$ if a definition of a intervenes between these occurrences along some path. This renaming affects the ability of an optimizer to identify all occurrences of the same expression in the original program [10]. Following [2], we solve this problem by representing an occurrence of an expression $a*b$ by an *expression variable* h^{a*b}. (We omit the superscript if it is obvious from the context.) Different version numbers of h^{a*b} are now awarded to occurrences of $a * b$ in which its operands have different version numbers. Thus $a_1 * b_1$ and $a_2 * b_1$ may be represented by the expression variables h_1^{a*b} and h_2^{a*b}. These version numbers are merged using a merge function analogous to ϕ. We use the symbol Φ for the merge function of expression variables to differentiate it from the merge function for program variables.

E-path_PRE contains the following steps:

1. Φ-insertion
2. Renaming
3. Computation of availability
4. Computation of anticipatability
5. Computation of e-paths
6. Computation of insertion points
7. Insertion
8. Elimination of redundant computations.

Φ-insertion and Renaming are used to compute an SSA form suitable for PRE. Availability and anticipatability properties are used to check for safety at the exit of a node. We compute e-paths by following the conditions mentioned in Defs. 2.1 and 2.2. Thus an e-path for an expression e starts with a node which has expression e available at its exit, ends with a node containing a locally anticipatable occurrence of e and contains intermediate nodes which are all empty with respect to expression e and have e available at their exit. To avoid tracing e-paths individually, we construct a special graph called *eliminatability graph* (G_l) for an expression e. G_l contains a subset of nodes and edges

in G such that each path in G_l is an e-path for expression e.

The basis for computation of insertion points is an edge $(y, x) \in G \ni x \in G_l$ but $(y, x) \notin G_l$. If x is an intermediate node or end node of an e-path then either node y or edge (y, x) would be considered for placement according to DPH–1 and DPH–2. The insertion step inserts the correct version of the expression at insertion points and detects extraneous Φ-functions. The last step replaces redundant computations of e by a temporary variable allocated to the correct version of h^e and changes the corresponding Φ-functions to ϕ-functions. The first four steps are applied only once to the entire program while the remaining steps are applied on an expression by expression basis. The following sections describe the design of individual steps of the algorithm and illustrate it with the help of an example.

3.1 Φ-Insertion

A Φ-function for expression e is inserted in each node which is in the iterated dominance frontier of a node containing an occurrence of e [14, 24] and in each node which contains a ϕ-function for an operand of e. In addition, we also insert a Φ-function in the entry node of a loop. This is done to simplify the computation of anticipatability.

3.2 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 at every Φ-function. We call the program form after Φ-insertion and renaming as PRESSA form to distinguish it from the SSA form.

In this step we also set two flags *available* and *has_real_use* for use in later steps. With every Φ-function $h_i \leftarrow \Phi(\ldots h_j \ldots)$ we associate flags $available_{h_i}$ = true and $has\_real\_use_{h_j}$ = false initially. 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', 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.

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. 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. While giving a version to an occurrence of $a * b$ we use the version numbers from the renaming stacks of a and b. We use the most recent version of h if the current versions of a and b match with the versions in the renaming stack of h, else we assign a new version to h. We use the following method to determine whether a Φ-operand is a real occurrence: From the renaming stack of h_j we check whether the version numbers of its operands match the version numbers at the top of their respective stacks and set $has\_real\_use_{h_j}$ = true if this is the case.

This step also computes a flag called *node_type* to indicate local properties of a node relevant for computation of availability. This flag takes the following values:

empty	node x is empty wrt 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.

3.3 Computation of Availability

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 occur-

rence 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. □

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. □

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 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. 2. The first pass traverses all Φ-functions. For each Φ function $h_j \leftarrow \Phi(\ldots)$ it sets availability to false if $available[h_j]$ = false. It then resets availability of all Φ-functions which have h_j as a Φ-operand and $has\_real\_use_{h_j}$ = false. 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

```
Algorithm Compute_availability(Droot)
{
    ∀ Φ-functions h_j ← Φ(...) ∋ available[h_j] = false
        ∀ Φ-functions h_i ← Φ(..., h_j, ...) ∋
        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-at-entry[x] ← false;
    ∀ expressions e
        If (x contains a Φ-function) then {
            Let h_i be the target of Φ-function;
            avail-at-entry[x] ← available[h_i];
        }
        Else {
            Let Idom_x be Immediate dominator of x;
            avail-at-entry[x] ← avail-at-exit[Idom_x];
        }
        If (node_type[x] is empty) then
            avail-at-exit[x] ← avail-at-entry[x];
        Else If ((node_type[x] is avail)
            or (node_type[x] is both)) then
                avail-at-exit[x] ← true;
            Else avail-at-exit[x] ← false;
}
Procedure Reset_availability(h_i)
{
    available[h_i] ← false;
        ∀ h_k ∋ h_k ← Φ(..., h_i, ...)
            If available[h_k] = true and has_real_use_{h_i}
            = false at h_k then
                Reset_availability(h_k);
}
```

Figure 2: Computation of availability

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.

Figure 3 shows the dominator tree for the program of Fig. 1. Figure 4 illustrates its PRESSA form, in which broken lines indicate def-use chains of h. 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. Hence $available_{h_2} = available_{h_3} = available_{h_4}$ = false. During availability computation this leads to *avail_at_exit* = 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

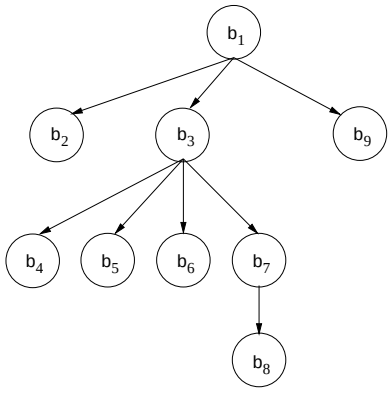

Figure 3: Dominator tree

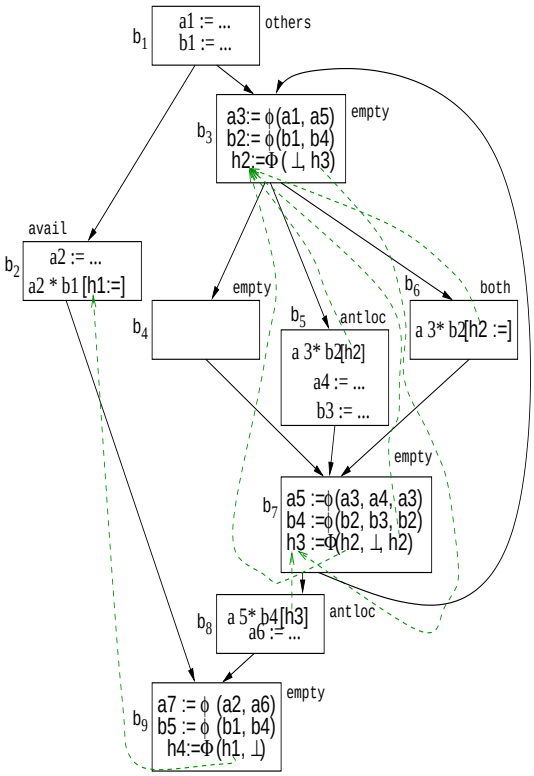

Figure 4: PRESSA form

located in node b_9 and $has\_real\_use_{h_2} = \text{true}$ for the Φ-function of h_3 located in node b_7.

3.4 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 [2] (it uses a property called down_safe which can be computed during renaming). 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 Φ insertion. Anticipatability at the exit of a node is computed from anticipatability at the entry of successor nodes. If anticipatability at the entry of a successor is not known, we recursively compute it before resuming the normal postorder traversal. Figure 5 contains the algorithm for computation of anticipatability. In the PRESSA form of Fig. 4, down_safe is computed at nodes b_9, b_7 and b_3. It is *false* in the first case but *true* in the other two cases. 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.

3.5 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. 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. 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 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

```
Algorithm Compute_Anticipatability(Droot)
{
    ∀ x
        ant-at-exit[x] ← ant-at-entry[x] ← true;
    ant-at-exit[exit] ← false;
    ant-at-entry[Droot] ← Anticipatability(Droot);
}
boolean Anticipatability(Droot)
{
    Let x be the current node in dom-tree visited
    in postorder;
    If (x is already visited) then
        Return(ant-at-entry[x]);
    For all expressions e do {
        If (x is not exit) then {
            If (∃ y ∈ succ(x) ∋
            ant-at-entry[y] = false) then
                ant-at-exit[x] ← false;
            Else {
                ∀ y ∈ succ(x) ∋ y is not visited
                    Ant_at_Successors(y);
                ant-at-exit[x] ←
                    Π_{y∈succ(x)} ant-at-entry[y];
            }
        }
        If (node_type[x] = empty) then
            ant-at-entry[x] ← ant-at-exit[x];
        Else
            If (node_type[x] = avail or others)
            then ant-at-entry[x] ← false;
    }
    Return(ant-at-entry[Droot]);
}
Procedure Ant_at_Successors(y)
{
        If (y is a loop entry node) then
            ant-at-exit[x] ← ant-at-exit[x] and
                down_safe(Φ-function in y);
        Else
            ant-at-exit[x] ← ant-at-exit[x]
                and Anticipatability(y);
}
```

Figure 5: Computation of anticipatability

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. After this step, the graph contains nodes which can be classified into start nodes, end nodes and intermediate nodes strictly according to Defs. 2.1 and 2.2. Note that G_l can be a multiple-entry multiple-exit graph which may contain one or more connected components. Figure 6 shows this algorithm. Note that a node x_l in G_l is a node which corresponds to node x in G. We use the superscript *in* or *out* if the node has been split while constructing G_l.

Lemma 3 $x_l \ldots z_l$ is a maximal path in G_l iff $x \ldots z$ 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. 2.1 and 2.2), nodes on the path $\alpha \equiv [x \ \ldots \ z]$ will be added to G_l and none of them will be removed as a useless node. Therefore there will be a corresponding path $\alpha_l \equiv [x_l \ldots z_l]$ 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 $[x_l \ldots z_l]$ is a maximal path in G_l.

(b) Only if part: $\alpha_l \equiv [x_l \ldots z_l]$ 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 $\alpha \equiv [x \ldots z]$ 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. □

Figure 7 illustrates G_l for the program of Fig. 4. 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} would also be added, however they would be removed during elimination of useless nodes.

3.6 Computation of Insertion Points

To perform insertion according to the code placement criteria DPH–1 and DPH–2, we need to iden-

```
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-at-exit[x] or ant-at-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);}
    Remove all isolated nodes in G_l;
    Uselessnodes_p ← {x_l ∈ G_l | node_type_x ≠
        (both or avail) and |pred(x)| = 0 }
    Uselessnodes_s ← {x_l ∈ G_l | node_type_x ≠
        (both or antloc) and |succ(x)| = 0 }
    ∀x ∈ Uselessnodes_p Remove_Uselessnodes_p(x);
    ∀x ∈ Uselessnodes_s Remove_Uselessnodes_s(x);
}
Procedure Append_node_to_paths(x)
{
    ∀ y ∈ pred(x) ∋ (visited[y] and node_type_y
    =empty/avail/both) Add_edge(y, x);
}
Procedure Append_successors(x)
{
    ∀y ∈ succ(x) ∋(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 ← succ(y); Remove y from G_l;
    ∀z ∈ S_y ∋ |pred(z)| = 0 Remove_Uselessnodes_p(z);
}
Procedure Remove_Uselessnodes_s(y)
{
    S_y ← pred(y); Remove y from G_l;
    ∀z ∈ S_y ∋ |succ(z)| = 0 Remove_Uselessnodes_s(z);
}
```

Figure 6: Computation of e-paths

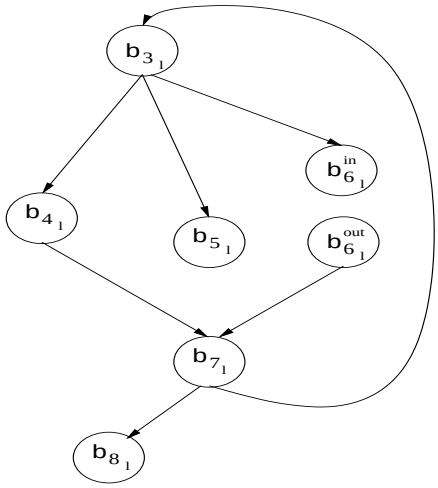

Figure 7: Illustration of e-path

```
Algorithm Compute_Insertion_Points(E-graph G_l)
{
    Insert = {};
    ∀ x_l ∈ G_l
        If | pred(x_l) | ≠ 0 then
            ∀ edges (y, x) ∈ G ∋
            ((y_l, x_l) or (y_l^out, x_l) ∉ G_l and
            avail_at_exit[y] = false) then
                Insert = Insert ⋃ {(y, x)};
}
```

Figure 8: Finding insertion points

tify 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. Figure 8 shows the algorithm. When applied to the G_l of Fig. 7, this algorithm computes Insert $= \{(b_1, b_3), (b_5, b_7)\}$.

3.7 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 [4]. 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.

Figure 9 contains the algorithm for insertion. Fig. 10 illustrates insertions performed by this algorithm. Insertions have been performed in node b_5 and along edge b_{1-3}. The SSA versions assigned to these insertions replace the $\perp$ operands in the Φ-functions situated in nodes b_7 and b_3, respectively.

3.8 Elimination of Redundant Computations

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 re-

```
Algorithm Insert(Droot : root of dom-tree)
{
    Let x be the current node of dom-tree
    in preorder;
    Collect_versions(x);
    If {(x, s) | s ∈ succ(x)} ⊆ Insert then
        Insert_in(x);
    Else ∀ (x, s) ∋ s ∈ succ(x)
        If ((x, s) ∈ 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 Φ-operand
        (if any) in the successor(s) of z by h_k;
    Insert an SSA version of the assignment
        h_k ← 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 Φ-function h_i ← Φ(...))
    and (avail-at-entry[x] = false or h_i has no
    use in the program)) then
        extraneousPhi[x] ← true;
    If ∃ an occurrence of e with version h_k in x ∋
    an extraneous Φ-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)
{
    ∀ assignments of the form h ← ...
    or an assignment to an operand opd of e in x
    in the order of their lexical occurrences in x
        Push the version given to h (or opd)
        onto the renaming stack of e (or opd);
}
```

Figure 9: Insertion of expressions

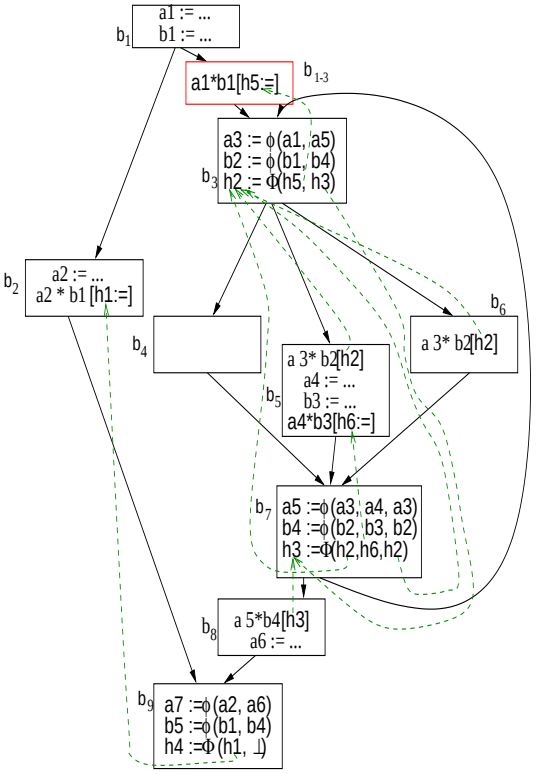

Figure 10: The PFG after insertion of computations.

placed 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. 11. Figure 12 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 Concluding Remarks

The advantages of E-path_PRE 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) expression-

```
Algorithm Eliminate_redundancy(PFG G)
{
    Remove all extraneous Φ-functions;
    Traverse each SSA def-use graph;
    Let node x contain a definition h_i ← ...
    If (the definition is h_l ← Φ(...)) then
        If (x is an intermediate or end node) then {
            Generate a unique t_i for h_l;
            Replace the Φ-function by t_i ← φ(...);
            Replace_use(h_l, t_i);
            If (t_i ← φ(...) 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 ← e by t_i ← e;
            Replace_use(h_l, t_i);
        }
    Remove all remaining Φ-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 ← Φ(... h_l ...) then {
            Generate a unique name t_g and replace
            the Φ function by t_g ← φ(... t_i ...);
            Replace_use(h_k, t_g);
            If (t_i ← φ(...) 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 ← φ(...) {
        Generate a unique t_l for h_k;
        If the definition of h_k is h_k ← Φ(...) then {
            Replace the Φ-function by t_l ← φ(...);
            Replace_use(h_k, t_l);
            If (∃ an h operand of t_l) then
                Replace_operand(t_l);
        }
        Else {
            Replace the definition h_k ← e by t_l ← e;
            Replace_use(h_k, t_l);
        }
    }
}
```

Figure 11: Elimination of redundancies

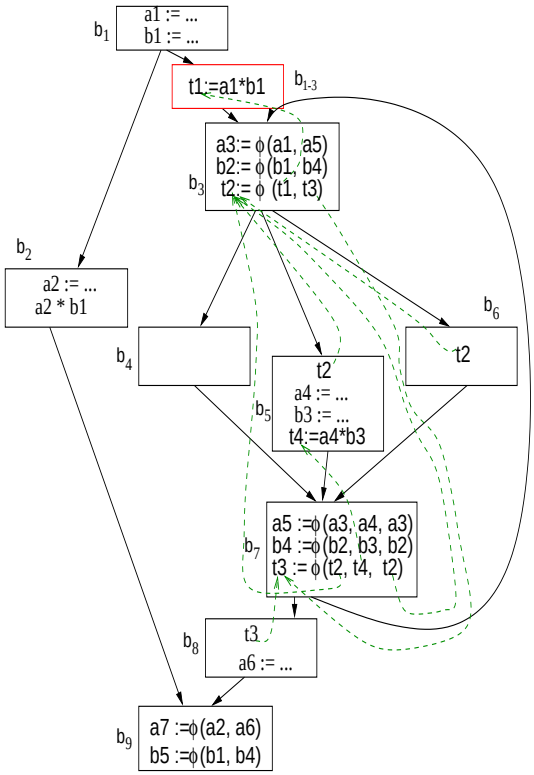

Figure 12: The optimized program

s [1, 23]. It also involves less work than SSAPRE, the SSA-based approach of [2]. 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.

Other advantages compared to the SSA based approach of [2] 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 [4, 9], 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 [2] 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 [11], 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.

References

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

[2] 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.

[3] 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.

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

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

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

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

[14] 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.

[15] 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.

[16] K. Kennedy. Safety of code movement. *International Journal of Computer Mathematics*, 3:112–130, 1972.

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

[18] 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.

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

[20] 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.

[21] 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.

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

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

[24] 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.