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

Ashish Kundu and D. M. Dhamdhere*
ashishk@cse.iith.ernet.in dmd@cse.iith.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 eliminability 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 $b_i$ such that some node along the path $[b_i \ldots b_k]$ is a successor of $b_i$.

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
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 eliminability 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.

**Eliminability 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 eliminable 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 eliminable occurrence of \( e \) in node \( b_k \) or a computation of \( e \) placed at the exit of \( b_k \) would be an eliminable occurrence of \( e \). Note that eliminability 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 eliminability 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 \( \{b_i\} \) and the set of synthetic nodes \( \{b_{1\ldots m}\} \) as defined by DPH-1 and DPH-2:

**[DPH-1]** A computation of \( e \) is placed at the exit of node \( b_i \) iff

a. \( e \) is not available at the exit of \( b_i \),

b. \( e \) is not eliminatable in \( b_i \), and

c. All paths starting at the exit of \( b_i \) have a prefix \([b_i \ldots b_k]\) such that \( b_k \) contains an eliminable occurrence of \( e \), and for all nodes \( b_j \) on the path \((b_i \ldots b_k)\), \( e \) 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 \( b_{1\ldots m} \) inserted on the edge \((b_i,b_{m})\) iff

a. \( e \) cannot be placed in \( b_i \) due to the violation of condition DPH-1(c),

b. \( e \) is neither available nor eliminatable at the exit of node \( b_i \), 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 eliminable 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 \( b_i \) (original node or synthetic node) to eliminate the eliminable occurrence of \( e \) in \( b_k \). Let \( b_s \) be a successor of \( b_i \) along the path \((b_i \ldots b_k)\). From criteria (c) of DPH-1 and DPH-2 it follows that \( e \) is eliminatable from \( b_s \). Hence
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 \( \phi \) 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 \( a \times b \) may be replaced by \( a_1 \times b_i \) and \( a_2 \times b_1 \) 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 \( a \times b \) by an expression variable \( h^{a \times b} \). (We omit the superscript if it is obvious from the context.) Different version numbers of \( h^{a \times b} \) are now awarded to occurrences of \( a \times b \) in which its operands have different version numbers. Thus \( a_1 \times b_1 \) and \( a_2 \times b_1 \) may be represented by the expression variables \( h_1^{a \times b} \) and \( h_2^{a \times b} \). These version numbers are merged using a merge function analogous to \( \phi \). We use the symbol \( \Phi \) 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. \( \Phi \)-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.

\( \Phi \)-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 **eliminability graph** \( G_{\Phi} \) for an expression \( e \). \( G_1 \) contains a subset of nodes and edges

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

The basis for computation of insertion points is an edge $(y, x) \in G \ni x \in G_i$ but $(y, x) \notin G_i$. 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 $\Phi$-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 $\Phi$-functions to $\phi$-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 $\Phi$-Insertion

A $\Phi$-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 $\phi$-function for an operand of $e$. In addition, we also insert a $\Phi$-function in the entry node of a loop. This is done to simplify the computation of anticipatabley.

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 $\Phi$-function. We call the program form after $\Phi$-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 $\Phi$-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 $\Phi$-function (henceforth called a $\Phi$-operand) represents the value of $e$ along some path $\beta$ reaching the $\Phi$-function. A $\Phi$-operand is assigned the special version $\bot$ if $\beta$ does not contain a $\Phi$-function for $e$ or an occurrence of $e$ following last definition(s) of its operand(s). This indicates that $e$ is not available along $\beta$. We set available$_{h_i} = false$ if a $\Phi$-function $h_i \leftarrow \Phi(\ldots)$ has such an operand. We say an operand $h_j$ corresponding to path $\beta$ in a $\Phi$-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 $\Phi$-function exists along $\beta$. Such a $\Phi$-operand indicates that value of $e$ is available along $\beta$.

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 $h_i$ else we assign a new version to $h$. We use the following method to determine whether a $\Phi$-operand is a real occurrence: From the renaming stack of $h_i$ 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 $\Phi$-function for expression $e$ and no dominator of $x$ contains either a $\Phi$-function or an occurrence of the expression $e$.

Proof : Consider a dominator dom$_x$ of $x$. Since dom$_x$ does not contain a $\Phi$-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 $\Phi$-function for $e$, which is also a contradiction. □

Lemma 2 If a node $x$ does not contain a $\Phi$-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 $\alpha$, then $\alpha$ contains a definition of some operand $v$ of $e$. This would lead to a $v$-function in $x$ which would give rise to a $\Phi$-function for $e$ in $x$, a contradiction. If availability is generated along $\alpha$, then there is an occurrence of $e$ along $\alpha$, which will lead to a $\Phi$-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 $\Phi$-function, then availability at its entry is same as availability at the exit of its immediate dominator. If a node contains a $\Phi$-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 $\Phi$-operand $h_j$ is itself the result of a $\Phi$-function, then availability transitively depends on availability at the $\Phi$-function $h_j \leftarrow \Phi(\ldots)$ — it would be false if availability is false at the $\Phi$-function of $h_j$. Finally, availability is true if has$_{realuse}_{h_j} = true$ for each $\Phi$-operand $h_j$.

The algorithm to compute availability consists of two passes as shown in Fig. 2. The first pass traverses all $\Phi$-functions. For each $\Phi$-function $h_j \leftarrow \Phi(\ldots)$ it sets availability to false if available$_{h_j} = false$. It then resets availability of all $\Phi$-functions which have $h_j$ as a $\Phi$-operand and has$_{realuse}_{h_j} = false$. 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 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 $h_i$ does not contain a $\Phi$-function, then avail$_{at\_entry}$ is simply avail$_{at\_exit}$ of its immediate dominator.

Figure 2: Computation of availability

Algorithm Compute availability(Droot) \{ 
\begin{align*}
& \forall \Phi\text{-functions } h_j \leftarrow \Phi(\ldots) \ni \text{available}[h_j] = false \\
& \forall \Phi\text{-functions } h_i \leftarrow \Phi(\ldots, h_j, \ldots) \ni \\
& \quad \text{available}[h_i] = true \text{ and has}_{\text{realuse}}_{h_i} = false \text{ at } h_k \\
& \text{Reset availability}(h_i); \\
& \text{Let } x \text{ be the current node visited in preorder;} \\
& \text{If } x \text{ is the entry node of PFG, then} \\
& \quad \text{avail}_{at\_entry}[x] \leftarrow false; \\
& \forall \text{expressions } e \\
& \text{If } (x \text{ contains a } \Phi\text{-function}) \text{ then } \{ \\
& \quad \text{Let } h_i \text{ be the target of } \Phi\text{-function;} \\
& \quad \text{avail}_{at\_entry}[x] \leftarrow \text{available}[h_i]; \\
& \} \\
& \text{Else } \{ \\
& \quad \text{Let } Idom_x \text{ be Immediate dominator of } x; \\
& \quad \text{avail}_{at\_entry}[x] \leftarrow \text{avail}_{at\_exit}[Idom_x]; \\
& \} \\
& \text{If } (\text{node\_type}[x] \text{ is empty}) \text{ then} \\
& \quad \text{avail}_{at\_exit}[x] \leftarrow \text{avail}_{at\_entry}[x]; \\
& \text{Else If } ((\text{node\_type}[x] \text{ is avail}) \\
& \quad \text{or (node\_type}[x] \text{ is both})) \text{ then} \\
& \quad \text{avail}_{at\_exit}[x] \leftarrow true; \\
& \text{Else avail}_{at\_exit}[x] \leftarrow false; \\
& \}\}

Procedure Reset availability(h_i) \{ 
\begin{align*}
& \text{available}[h_i] \leftarrow false; \\
& \forall h_k \ni h_k \leftarrow \Phi(\ldots, h_i, \ldots) \\
& \quad \text{If available}[h_k] = true \text{ and has}_{\text{realuse}}_{h_i} = false \text{ at } h_k \\
& \quad \text{Reset availability}(h_k); \\
& \}\}

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 $\downarrow$ is assigned to a $\ast b$ at the exit of blocks $b_1$, $b_2$, and $b_8$ because $a \ast b$ is not available at their exit. Hence available$_{b_2} = \text{available}_{b_3} = \text{available}_{b_4} = false$. During availability computation this leads to avail$_{at\_exit} = false$ for all nodes in $G$ except for nodes $b_2$ and $b_8$. Note that has$_{realuse}_{h_1} = true$ for the $\Phi$-function of $h_1$. 

5
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 \( \text{node}\_\text{type} \) attribute computed during \( \Phi \) 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, \( \text{down}_\text{safe} \) is computed at nodes \( b_9, b_7 \) and \( b_3 \). It is \( \text{false} \) in the first case but \( \text{true} \) in the other two cases. Anticipatability at entry of node \( b_9 \) is true since the node is \( \text{antloc} \). Propagation of these values yields \( \Theta = \text{true} \) 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 \( G_I \) such that each path in \( G_I \) is an e-path in \( G \). Nodes and edges in \( G_I \) are the nodes and edges in \( G \), except for one difference — some nodes of \( G \) are split into two nodes of \( G_I \). This is done to facilitate construction of \( G_I \). Node splitting is motivated by the following considerations: An e-path for expression \( e \) starts with a node whose \( \text{node}\_\text{type} \) is \( \text{avail} \) or \( \text{both} \) (we will call this a \( \text{start} \) node), ends with a node whose \( \text{node}\_\text{type} \) is \( \text{antloc} \) or \( \text{both} \) (an \( \text{end} \) node), and all intermediate nodes \( x \) in it have \( \text{node}\_\text{type}_x = \text{empty} \) and \( \text{avail}_x = \text{true} \) or \( \text{ant}_x = \text{true} \). It may be noted that a node \( x \) with \( \text{node}\_\text{type} = \text{both} \) 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 \( x^\text{in} \) and \( x^\text{out} \), which represent the parts containing the entry and exit of node \( x \), respectively. All in-edges of node \( x \) become in-edges of \( x^\text{in} \) and all out-edges of node \( x \) become out-edges of \( x^\text{out} \). Nodes \( x^\text{in} \) and \( x^\text{out} \) do not have any out-edges and in-edges, respectively. As described at the start of this section, the motivation for building \( G_I \) 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 \( G_I \). 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 \( \text{node}\_\text{type}_x = \text{empty/antloc/both} \)) and a predecessor \( p \) of \( x \) can be a start or intermediate node of an e-path (indicated

![Figure 3: Dominator tree](image)

![Figure 4: PRESSA form](image)

located in node \( b_3 \) and \( \text{has}_\text{real}_\text{use}_x \) is true for the \( \Phi \)-function of \( b_3 \) located in node \( b_7 \).

### 3.4 Computation of Anticipatability

Computation of anticipatability requires two passes. The first pass computes anticipatability at each \( \Phi \)-function. This pass is analogous to the Down-Safety pass of SSAPRE [2] (it uses a property called down-safe which can be computed during renaming). The second pass computes anticipatability at...
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] ← Πy∈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(∅-function in y);
    Else
        ant-at-exit[x] ← ant-at-exit[x]
        and Anticipatability(y);
}

by node-type = empty/avail/both) then edge (p, x) is added to Gt. Similarly an edge (x, s) is added for a successor s if node-type,x = empty/avail/both, and node-type,s = empty/and/loc/both. It now splits x if node-type,x = both as described earlier. This procedure is conservative, hence some nodes created in Gt may not belong to an e-path. Hence we prune the graph by removing all useless nodes from Gt, 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 Gt 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 xι in Gt 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 Gt.

Lemma 3 xι...zι is a maximal path in Gt iff x...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 α ≡ [x...z] will be added to Gt and none of them will be removed as a useless node. Therefore there will be a corresponding path αⅰ ≡ [xι...zι] in Gt. Since x and y are start node and end node, respectively, |pred(x)| = 0 and |succ(y)| = 0. Hence [xι...zι] is a maximal path in Gt.

(b) Only if part: αⅰ ≡ [xι...zι] is a maximal path in Gt. An edge (p, q) is added to Gt only if an edge (p,q) exists in G. Hence there exists α ≡ [x...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 Gt, a contradiction.

Figure 7 illustrates Gt for the program of Fig. 4. Node b8 is split into nodes b8′ and b8″ as node-type = both. Nodes b7, b1 and b4 are added to Gt because they are empty. Nodes b5 and b9 are added because they are antloc. Nodes b2 and b3 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)
{
  Travers the dominator tree in preorder;
  Set visited[y] of current node x to true;
  If \( (x \neq \text{entry or exit node}) \)
  and \( \text{node\_type}[x] = \text{both} \) then
    Mark node x for splitting;
    If \( \text{node\_type}[x] = \text{empty} \) and
    \( \text{at-end} \) or \( \text{at-end}[x] \) then
      Append \_node\_to\_paths(x);
      Append \_successors(x);
    Else if \( \text{node\_type}[x] = \text{at-end} \) then
      Append \_node\_to\_paths(x);
    Else if \( \text{node\_type}[x] = \text{avail} \) then
      Append \_successors(x);
    Else if \( \text{node\_type}[x] = \text{both} \) then
      Append \_node\_to\_paths(x);
      Append \_successors(x);
    Remove all isolated nodes in Gt;
    Uselessnodes\_p \leftarrow \{ x \in Gt \mid \text{node\_type}_x \neq \text{both or avail} \} \}
    Uselessnodes\_s \leftarrow \{ x \in Gt \mid \text{node\_type}_x \neq \text{both or antile} \} \}
    \forall x \in \text{Uselessnodes}_p \text{ Remove} \_\text{Uselessnodes}_p(x);
    \forall x \in \text{Uselessnodes}_s \text{ Remove} \_\text{Uselessnodes}_s(x);
}
Procedure \_node\_to\_paths(x)
{
  \forall y \in \text{pred}(x) \exists (\text{visited}[y] \text{ and } \text{node\_type}_y = \text{empty/avail/both}) \text{ Add} \_\text{edge}(y,x);
}
Procedure \_successors(x)
{
  \forall y \in \text{succ}(x) \exists (\text{visited}[y] \text{ and } \text{node\_type}_y = \text{empty/antile/both}) \text{ Add} \_\text{edge}(x,y);
}
Procedure \_\text{edge}(x,y)
{
  \text{If (x has been marked for splitting) then}
    \text{Create node x^\text{new} in Gt if not present;}
  \text{Else create node x in Gt if not present;}
  \text{If (y has been marked for splitting) then}
    \text{Create node y^\text{new} in Gt if not present;}
  \text{Else create node y in Gt if not present;}
  \text{Add a directed edge (x^*, y^*) where}
    \text{:x^* is 'out', or blank and :y^* is 'in' or blank;}
}
Procedure \_\text{Remove} \_\text{Uselessnodes}_p(y)
{
  \text{S_p} \leftarrow \text{succ}(y); \text{Remove y from Gt};
  \forall z \in S_p \exists |\text{pred}(z)| = 0 \text{ Remove} \_\text{Uselessnodes}_p(z);
}
Procedure \_\text{Remove} \_\text{Uselessnodes}_s(y)
{
  \text{S_s} \leftarrow \text{pred}(y); \text{Remove y from Gt};
  \forall z \in S_s \exists |\text{succ}(z)| = 0 \text{ Remove} \_\text{Uselessnodes}_s(z);
}

Figure 7: Illustration of e-path

Algorithm Compute\_\text{Insertion} \_\text{Points}(E-graph Gt)
{
  Insert = \{ \};
  \forall x_i \in Gt
    If \text{ |pred(x)| = 0} then
      \forall edges \((y,x) \in Gt\) \exists
        \((y^\text{new}, x)\) or \((y, x^\text{new})\) \text{ and}
        \text{at-end}(y) \text{ false then}
        Insert = Insert \cup \{ (y, x) \};
}

Figure 8: Finding insertion points

ify 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 Gt 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 Gt 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
\(\Phi\)-functions. The previous step has marked edges
(y, x) for insertion where x is a node in some e-path w . . . x . . . 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 . . . 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_i = Φ(...) which is followed by an occurrence of h_i along some path. If this occurrence of h_i is eliminatable, then an appropriate version of t_i, say t_i', would be allocated to h_i and the Φ-function would be replaced by t_i' ← φ(...) in the next step. If no occurrence of h_i is eliminatable, then the Φ-function for h_i 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_i ← . . . along a path starting on the Φ-function. Uses of h_i 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_y ← e is inserted in node y, or along edge (y, x), the name h_y 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 . . . . The SSA versions assigned to these insertions replace the Φ-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_i to every version of h, say h_i'. The definition h_i' ← e in the start node of an e-path and in a node marked for insertion is replaced by t_i ← e. Each occurrence h_k ← Φ(...) is re-

Algorithm Insert(Droot : root of dom-tree)
{
  Let x be the current node of dom-tree in preorder;
  Collect_versions(x);
  If \( \{ (x, s) \mid s \in \text{succ}(x) \} \subseteq \text{Insert} \)
  Insert.In(x);
  Else \( \forall (x, s) \ni s \in \text{succ}(x) \)
  If \( (x, s) \in \text{Insert} \) then
    Insert a synthetic block b_{z→} to
    split the edge \( (x, s) \);
    Insert.In(b_{z→});
  }
  DetecExtraneousPhi(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(x)
{
  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 DetecExtraneousPhi(x)
{
  If \((x \text{ is not in an e-path}) \) and
  \((x \text{ contains a Φ-function } h_i ← \Phi(...) ) \)
  and \((\text{avail-at-entry}[x] \text{ = false or } h_i \text{ has no use in the program}) \)
  then
    extraneousPhi[x] ← true;
  If \( \exists \) an occurrence of e with version h_k in x \( \exists \)
  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)
{
  \( \forall \) assignments of the form h ← . . .
  or an assignment to an operand op of e in x
  in the order of their lexical occurrences in x
  Push the version given to h (or op)
  onto the renaming stack of e (or op);
}

Figure 9: Insertion of expressions
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 \( h_i \leftarrow \Phi(\ldots) \)) then 
       If (\( x \) is an intermediate or end node) then 
          Generate a unique \( t_i \) for \( h_i \); 
          Replace the \( \Phi \)-function by \( t_i \leftarrow \phi(\ldots) \); 
          \( \text{Replace}_{\text{use}}(h_i, t_i) \);
          If (\( t_i \leftarrow \phi(\ldots) \) has an \( h \) operand) then 
             \( \text{Replace}_{\text{operand}}(t_i) \);
       }
    Else If ((the definition of \( h_i \) is an inserted 
              occurrence or \( x \) is a start node) or (\( x \) has a 
              definition and a real occurrence of \( h_i \))) then 
       Generate a unique \( t_i \) for \( h_i \); 
       Replace the definition \( h_i \leftarrow e \) by \( t_i \leftarrow e \); 
       \( \text{Replace}_{\text{use}}(h_i, t_i) \);
    }
    Remove all remaining \( \Phi \)-functions; 
    Remove all remaining \( h \) versions, without 
    removing the occurrences of the expressions; 
}
Procedure Replace_{\text{use}}(h_i, t_i)
{
    For each entry \( y \) in the def-use chain of \( h_i \)
    If use is of the form \( h_k \leftarrow \Phi(\ldots h_i; \ldots) \) then 
        Generate a unique name \( t_g \) and replace 
        the \( \Phi \)-function by \( t_g \leftarrow \phi(\ldots t_i; \ldots) \); 
        \( \text{Replace}_{\text{use}}(h_k, t_g) \);
        If (\( t_i \leftarrow \phi(\ldots) \) contains an \( h \) operand) 
           then \( \text{Replace}_{\text{operand}}(t_g) \);
    }
    Else 
    Replace the use by \( t_i \);
}
Procedure Replace_{\text{operand}}(t_i)
{
    For each \( h \)-operand \( h_k \) of \( t_i \) \( \leftarrow \phi(\ldots) \)
    Generate a unique \( t_i \) for \( h_k \); 
    If the definition of \( h_k \) is \( h_k \leftarrow \Phi(\ldots) \) then 
        Replace the \( \Phi \)-function by \( t_i \leftarrow \phi(\ldots) \); 
        \( \text{Replace}_{\text{use}}(h_k, t_i) \);
        If (\( \exists \) an \( h \) operand of \( t_i \) then 
          \( \text{Replace}_{\text{operand}}(t_i) \);
    }
    Else 
    Replace the definition \( h_k \leftarrow e \) by \( t_i \leftarrow e \); 
    \( \text{Replace}_{\text{use}}(h_k, t_i) \);
}

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-
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


