Partial redundancy elimination: A simple, pragmatic, and provably correct algorithm

V.K.Paleri, Y.N.Srikant, and P.Shankar

Department of Computer Science and Automation, Indian Institute of Science
Bangalore-560 012, India

Abstract

We propose a new algorithm for partial redundancy elimination based on the new concepts of *safe partial availability* and *safe partial anticipability*. These new concepts are derived by the integration of the notion of *safety* into the definitions of partial availability and partial anticipability. The algorithm works on flow graphs whose nodes are basic blocks. It is both computationally and lifetime optimal and requires four unidirectional analyzes. The most important feature of the algorithm is its simplicity; the algorithm evolves naturally from the new concept of safe partial availability.

Key Words: Computational optimality, data-flow analysis, flow graphs, life-time optimality, partial redundancy elimination.

1 Introduction

A computation which is performed twice in a certain path in the program is said to be partially redundant. Partial redundancy elimination involves the insertion and deletion of computations in the program in such a way that after the transformation the program contains no more - in general, fewer - occurrences of such computations. In order to preserve the semantics of the original program, the insertions of computations corresponding to the transformation must be *safe*, i.e., it must not introduce computations of new values on any path in the program.

Morel and Renvoise showed that global common subexpression elimination and loop invariant code motion are special cases of partial redundancy elimination [12]. They viewed partial redundancy elimination as a program flow analysis problem in which the points of insertion and deletion of computations are determined by solving data-flow equations. The technique is based on a purely Boolean approach and hence permits simultaneous treatment of all expressions of a program using bit-vectors. The algorithm does not require detailed control flow analysis.

Morel and Renvoise's algorithm does not eliminate all partial redundancies that exist in a program. Taking the issue of *safety* into account Dhamdhere provided a solution which eliminated all partial redundancies, using the concept of edge placement [4, 5]. The solution given by Morel and Renvoise also has the problem of redundant code motion - code movement without any execution time gains. This problem was addressed in several papers [2, 4, 5], which reduced but could not entirely prevent redundant code motion.

The algorithm given by Morel and Renvoise involves bidirectional data-flow analysis. Bidirectional analyzes are, in general, conceptually and computationally more complex than unidirectional ones [5, 6, 7]. It was shown that the transformation can also be solved as a unidirectional problem [3, 4, 8].

All the above solutions for partial redundancy elimination have one or more of the problems of redundant code motion, unremoved redundancies, or limited applicability due to reducibility restriction of the flow graph. Knoop, Rüthing, and Steffen proposed a computationally optimal algorithm, composed of unidirectional analyzes, for structurally unrestricted flow graphs, with no redundant code motion [10]. A variant of this algorithm was given by Dreshler and Stadel in [9]. Knoop et.al., later presented an algorithm [11] which works on flow graphs whose nodes are basic blocks as against nodes with single statements in their earlier work [10].

Here, we propose a new algorithm for partial redundancy elimination. A preliminary version of this algorithm, with each node of the control flow graph as single statement, can be found in our earlier work [13]. The algorithm is based on the new concepts of *safe partial availability* and *safe partial anticipability*, concepts derived by the integration of the notion of *safety* into the definitions of partial availability and partial anticipability. As the concept of partial redundancy is based on partial availability, we have the concept of *safe partial redundancy* based on safe partial availability. Using safe partial redundancy we have the following points as the basis for the algorithm:

- Every safe partially redundant computation offers scope for redundancy elimination.
- Any safe partially redundant computation at a point can be made totally redundant by insertion of new computations at proper points, without changing the semantics of the program.
- Computation of any expression that is totally redundant can be replaced by a copy rule.

With the above points as basis, we can make the following conclusion:

- After the transformation, no expression is recomputed at a point if its value is *available* from previous computations.

The algorithm requires four unidirectional data-flow analyzes for the computation of availability, anticipability, safe partial availability, and safe partial anticipability. It is computationally and lifetime optimal; after the transformation, the number of chosen computations on each path is the minimum and the live ranges of the new temporaries introduced are also the minimum. The algorithm is practical as it works on flow graphs whose nodes are basic blocks. In comparison with its predecessor [11], our algorithm does not require the edge splitting transformation to be done before its application - edge splitting is done only at places where insertion of computation is necessary. The most important feature of the algorithm is its conceptual simplicity.

2 The Basic Concept

Here, we give an informal description of the basic idea behind our algorithm. We say an expression is *available* at a point if it has been computed along all paths reaching this point with no changes to its operands since the computation. An expression is said to be *anticipable* at a point if every path from this point has a computation of that expression with no changes to its operands in between. We say a point is *safe* for an expression if it is either available or anticipable at that point. *Partial availability* and *partial anticipability* are weaker properties with the requirement of a computation along "at least one path" as against "all paths" in the case of availability and anticipability.

Safe partial availability(or anticipability) at a point differs from partial availability(or anticipability) in that it requires all points on the path along which the computation is partially

available(or anticipable) to be safe. In the example given in Figure 1(a), partial availability of the expression $a + b$ at the entry of node 4 is true but safe partial availability at that point is false, because the entry and exit points of node 3 are not safe. In Figure 1(b), safe partial availability at the entry of node 4 is true. We say a computation is *safe partially redundant* in a node, if it is locally anticipable and is safe partially available at the entry of the node. In Figure 1(b), the computation in node 4 is safe partially redundant.

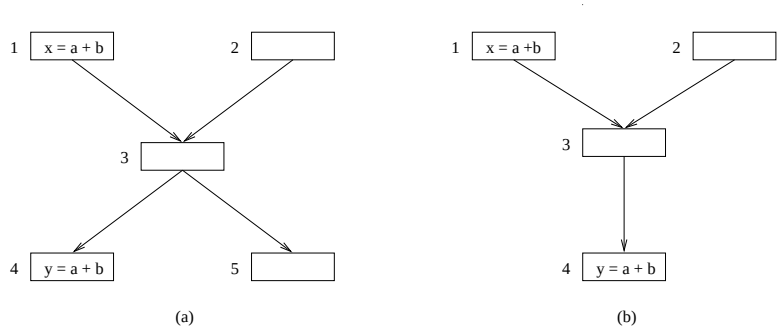

Figure 1: Safe partial availability

The algorithm assumes that all local redundancies are already eliminated by means of some standard techniques for common subexpression elimination on basic blocks [1]. After the removal of all local redundancies within a basic block i, we note that there exists at most one computation before the first modification, which we call the *first computation*, denoted by $FIRST_i$, and at most one computation before the first modification starting from bottom, which we call the *last computation*, denoted by $LAST_i$. All other computations within the basic block appear in between two modifications and hence are irrelevant for our algorithm. We call the *first* and *last* computations as *candidate computations* - computations which are relevant to the algorithm. Note that the first and last computations coincide when a single computation in a basic block has no modification to its operands in the block.

The basis of the algorithm is to identify safe partially redundant computations and make them totally redundant by the insertion of new computations at proper points. The totally redundant computations after the insertions are then replaced. If $a + b$ is the expression of interest then by insertion we mean insertion of the computation $h = a + b$, where h is a new variable; replacement means substitution of a computation, like $x = a + b$, by $x = h$.

Given a control flow graph we first compute availability and anticipability at the entry and exit points of all nodes in the graph by two iterative analyzes. From availability and anticipability we compute safety - a simple computation, not an iterative analysis - at all points. After computing safety, we compute safe partial availability and safe partial anticipability at the entry and exit points of all nodes, which require another two iterative analyzes. We then mark all points which satisfy both safe partial availability and safe partial anticipability. Now, consider the paths formed by connecting all the adjacent points which are marked. We observe that the required points of insertion for the transformation are the ones just before the *last computation* in nodes corresponding to the starting points of such paths and also the edges that enter *junction nodes* on these paths. The computations to be replaced are the ones appearing on these paths; for a path, only the *last computation* in the node corresponding to the starting point and only the *first computation* in the node corresponding to the end point are considered to be on the path.

Consider the example given in Figure 2(a). We have marked all points satisfying safe partial availability and safe partial anticipability by small circles. Path $< 1, 3, 4 >$ connects all adjacent marked points. Based on the above observation, we see that the point just before

the *last computation* in node 1 and also edge (2, 3) are the points of insertion and the *last computation* in node 1 and the *first computation* in node 4 are the computations to be replaced. The graph after the transformation is shown in Figure 2(b).

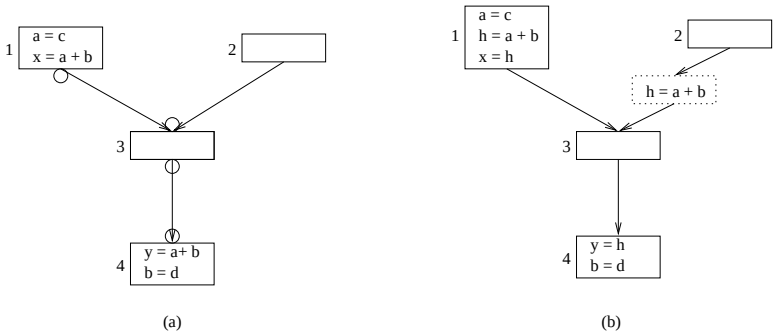

Figure 2: Partial redundancy elimination (a) Before transformation (b) After transformation

3 Notations and Definitions

3.1 Flow Graph

We represent a program as a directed flow graph $G = (N, E, s, e)$, where N is the set of nodes of the flow graph, E the set of edges of the flow graph, s the unique entry node with no predecessors, and e the unique exit node without any successors. Nodes $n \in N$ represent basic blocks consisting of a linear sequence of three-address statements. Arithmetic statements are of the form $v := expr$, where v is a variable and $expr$ is a simple expression, like $a + b$, built of variables, constants, and operators, having at most one operator. Both nodes s and e are assumed to be empty. Edges $(i, j) \in E$ represent the control flow from node i to node j in the flow graph. Every node $n \in N$ is assumed to lie on a path from s to e. The sets $succ(n) = \{m|(n, m) \in E\}$ and $pred(n) = \{m|(m, n) \in E\}$ denote the set of all immediate successors and immediate predecessors, respectively, of node n. A finite path of G is a sequence $< n_1, \ldots, n_k >$ of nodes such that $n_{i+1} \in \text{succ}(n_i)$ for all $1 \leq i < k$. A path from node i to node j is denoted by $p[i, j]$. If i or j is excluded from the path we will write it as $p]i, j]$ or $p[i, j[$, respectively. The set of all finite paths of G leading from a node i to a node j is denoted by $P[i, j]$.

3.2 Boolean Properties

We use the same terminology used by Morel and Renvoise [12]. For each expression and each node Boolean properties are defined. Some of these properties depend only on the statement in the node and are termed local. Other properties which depend on statements beyond a node are termed global. We define the Boolean properties and develop our algorithm for an arbitrary and fixed expression, since a global algorithm dealing with all expressions simultaneously is the independent combination of all of them, which can be realized using bit-vectors.

3.2.1 Local Properties

We associate the local properties, transparency, availability, and anticipability with a node. An expression is said to be *transparent* in a node i, denoted by $TRANSP_i$, if its operands are not modified by the execution of the statements in node i. We say an expression is *locally available*

in a node i, and denote it by $COMP_i$, if there is at least one computation of the expression in the node and if the statements appearing in the node including and after the last computation of the expression do not modify its operands. An expression is said to be *locally anticipable* in a node i, denoted by $ANTLOC_i$, if there is at least one computation of the expression in the node and if the statements appearing in the node before the first computation of the expression do not modify its operands.

3.2.2 Global Properties

The global properties, availability, anticipability, safe partial availability, and safe partial anticipability are of interest to us and we use $AVIN_i$, $ANTIN_i$, $SPAVIN_i$, and $SPANTIN_i$ to denote these properties (respectively), of an expression at the entry of node i. Similarly, $AVOUT_i$, $ANTOUT_i$, $SPAVOUT_i$, and $SPANTOUT_i$ are used to denote the same properties at the exit of node i.

We use $SAFEIN_i$ and $SAFEOUT_i$ to denote the fact that it is safe to insert a computation at the entry and exit, respectively, of node i. We say a path $p[m, n]$, from point m to point n in the flow graph, is *safe* if every point on the path is safe and denote it by $SAFE_{[m,n]}$. Also, we say a path $p[i, j]$, where i and j are nodes, is transparent if every node on the path is transparent and denote it by $TRANSP_{[i,j]}$.

The relation between global and local properties for all nodes of the graph are expressed in terms of systems of Boolean equations. Boolean conjunctions are denoted by . and $\prod$, disjunctions by + and $\sum$, and Boolean negation by $\neg$.

Availability. An expression is said to be available at a point p if every path from the entry node s to p contains a computation of that expression, and after the last such computation prior to reaching p there are no modifications to its operands.

An expression is available at the entry of a node if it is available on exit from each predecessor of the node. An expression is available at the exit of a node if it is locally available or if it is available at the entry of the node and transparent in this node, i.e.,

$$AVIN_i = \begin{cases} FALSE & \text{if } i = s \\ \prod_{j \in pred(i)} AVOUT_j & \text{otherwise} \end{cases}$$

$$AVOUT_i = COMP_i + AVIN_i.TRANSP_i$$

Anticipability. An expression is said to be anticipable at a point p if every path from p to the exit node e contains a computation of that expression, and after p prior to reaching the first such computation there are no modifications to its operands.

An expression is anticipable at the exit of a node if it is anticipable at the entry of each successor of the node. An expression is anticipable at the entry of a node if it is locally anticipable or it is anticipable at the exit of the node and transparent in this node, i.e.,

$$ANTOUT_i = \begin{cases} FALSE & \text{if } i = e \\ \prod_{j \in succ(i)} ANTIN_j & \text{otherwise} \end{cases}$$

$$ANTIN_i = ANTLOC_i + ANTOUT_i.TRANSP_i$$

Safety. A point p is considered to be safe for an expression if the insertion of a computation of that expression at p does not introduce a new value on any path through p. Alternatively, a point p is safe if the expression is either available or anticipable at that point. The points of

interest to us are the entry and exit of nodes, i.e.,

$$\begin{aligned} SAFEIN_i &= AVIN_i + ANTIN_i \\ SAFEOUT_i &= AVOUT_i + ANTOUT_i \end{aligned}$$

Safe Partial Availability. We say an expression is safe partially available at a point n, if there is at least one path from the entry node s to n which contains a computation of that expression, and after the last such computation on this path prior to reaching n, say at node m, there are no modifications to its operands, the path from the exit of node m to n being safe.

An expression is safe partially available at the entry of a node if the entry point of the node is safe and the expression is safe partially available on exit from at least one predecessor of the node. An expression is safe partially available at the exit of a node if the exit point of the node is safe and the expression is locally available or is safe partially available at the entry of the node and transparent in this node.

$$SPAVIN_i = \begin{cases} FALSE & \text{if } i = s \text{ or } \neg SAFEIN_i \\ \sum_{j \in pred(i)} SPAVOUT_j & \text{otherwise} \end{cases}$$

$$SPAVOUT_i = \begin{cases} FALSE & \text{if } \neg SAFEOUT_i \\ COMP_i + SPAVIN_i.TRANSP_i & \text{otherwise} \end{cases}$$

Safe Partial Anticipability. We say an expression is safe partially anticipable at a point m, if there is at least one path from m to the exit node e which contains a computation of that expression, and after m prior to reaching the first such computation on this path, say at node n, there are no modifications of its operands, the path from m to the entry of node n being safe.

An expression is safe partially anticipable at the exit of a node if the exit point of the node is safe and the expression is safe partially anticipable at the entry of at least one successor of the node. An expression is safe partially anticipable at the entry of a node if the entry point of the node is safe and the expression is locally anticipable or is safe partially anticipable at the exit of the node and transparent in this node.

$$SPANTOUT_i = \begin{cases} FALSE & \text{if } i = e \text{ or } \neg SAFEOUT_i \\ \sum_{j \in succ(i)} SPANTIN_j & \text{otherwise} \end{cases}$$

$$SPANTIN_i = \begin{cases} FALSE & \text{if } \neg SAFEIN_i \\ ANTLOC_i + SPANTOUT_i.TRANSP_i & \text{otherwise} \end{cases}$$

Such systems of Boolean equations are usually solved using an iterative process, which can yield several solutions depending upon the initialization of the unknowns. For the Boolean systems above, the required solution is the largest one for the system involving the conjunction operator $\prod$ and the initialization value is TRUE for all the unknowns in this case. For the systems involving the disjunction operator $\sum$ the required solution is the smallest one and the initialization value is FALSE for all unknowns.

Safe Partial Redundancy. There are only two computations relevant to the algorithm in a basic block i - $FIRST_i$ and $LAST_i$. In a node i, the computation $FIRST_i$ - which implies $ANTLOC_i$ - is said to be safe partially redundant, denoted by $SPREDUND_{i_f}$, if the computation is also safe partially available at the entry of the node, i.e.,

$$SPREDUND_{i_f} = ANTLOC_i.SPAVIN_i$$

Note that the computation $LAST_i$ in a node i, when it is distinct from $FIRST_i$, cannot be safe partially redundant.

Total Redundancy. In a node i, the computation $FIRST_i$ - which implies $ANTLOC_i$ - is said to be totally redundant - or simply, redundant - denoted by $REDUND_{i_f}$, if the computation is also available at the entry of the node, i.e.,

$$REDUND_{i_f} = ANTLOC_i.AVIN_i$$

The computation $LAST_i$ in a node i is said to be redundant, denoted by $REDUND_{i_l}$, if the computation is available at a point, say p, just before it, i.e.,

$$REDUND_{i_l} = COMP_i.AV_p, \text{where } p \text{ is the point just before } LAST_i.$$

Note that AV_p denotes availability at a point p. Similarly, we denote anticipability at a point p as ANT_p.

Isolatedness. A computation is said to be isolated if it is neither safe partially available nor safe partially anticipable at that point. We denote the isolatedness of $FIRST_i$ and $LAST_i$, in a node i, by $ISOLATED_{i_f}$ and $ISOLATED_{i_l}$, respectively, i.e.,

$$ISOLATED_{i_f} = ANTLOC_i.\neg SPAVIN_i.\neg(TRANSP_i.SPANTOUT_i)$$

$$ISOLATED_{i_l} = COMP_i.\neg SPANTOUT_i.\neg(TRANSP_i.SPAVIN_i)$$

4 The Algorithm

The algorithm introduces new computations of the expression at points of the program chosen in such a way that the safe partially redundant computations become totally redundant. As in [11], our algorithm introduces a new auxiliary variable h for the expression concerned, inserts assignments of the form $h := expr$ at some program points, and replaces some of the candidate computations of the expression by h, to achieve the transformation. These points of insertions and replacements are computed by the algorithm.

We observe that for an optimal solution the points of insertion must be either just before the *last computation* in a basic block or on an edge in the flow graph, which are denoted by $INSERT_i$ and $INSERT_{(i,j)}$, respectively. There are two candidates for replacement in a basic block - the *first* and the *last* computations - and we denote the replacements of them by $REPLACE_{i_f}$ and $REPLACE_{i_l}$, respectively.

The steps of the algorithm are as follows:

1. Compute AVIN/AVOUT and ANTIN/ANTOUT for all nodes.
2. Compute SAFEIN/SAFEOUT for all nodes.
3. Compute SPAVIN/SPAVOUT and SPANTIN/SPANTOUT for all nodes.
4. Compute points of insertion and replacement $INSERT_i$, $INSERT_{(i,j)}$, $REPLACE_{i_f}$, and $REPLACE_{i_l}$.

The points of insertions and replacements are computed using the following equations:

$$\begin{aligned}
INSERT_i &= COMP_i.SPANTOUT_i.(\neg TRANSP_i + \neg SPAVIN_i) \\
INSERT_{(i,j)} &= \neg SPAVOUT_i.SPAVIN_j.SPANTIN_j \\
REPLACE_{i_f} &= ANTLOC_i.(SPAVIN_i + TRANSP_i.SPANTOUT_i) \\
REPLACE_{i_l} &= COMP_i.(SPANTOUT_i + TRANSP_i.SPAVIN_i)
\end{aligned}$$

The algorithm requires four unidirectional analyzes for the computation of availability, anticipability, safe partial availability, and safe partial anticipability. It does not require the edge splitting transformation before the application of the algorithm as in [11]. Edge splittings are done only at places where it is necessary. The algorithm is computationally and lifetime optimal.

5 Correctness and Optimality of the Algorithm

In this section, we give the proofs for the correctness, computational optimality, and lifetime optimality of the algorithm.

5.1 Correctness

Here, we prove that the algorithm performs partial redundancy elimination correctly.

Lemma 1 *All insertions of computations corresponding to the transformation are done at safe points.*

Proof: For a point p, we have, $SAFE_p = AV_p + ANT_p$. We consider an edge as a point of insertion and denote the safety on edge (i, j) by $SAFE_{(i,j)}$. Similarly, anticipability on edge (i, j) is denoted by $ANT_{(i,j)}$.
Let us consider the two cases of insertions:
Case(i): $INSERT_i$.
$INSERT_i$ inserts the computation at a point, say p, just before $LAST_i$. We show that point p is safe.

$$\begin{aligned} INSERT_i &= COMP_i.SPANTOUT_i.(\neg TRANSPi + \neg SPAVINi) \\ &\Rightarrow COMP_i \\ &\Rightarrow ANT_p \quad [\text{The computation corresponding to } COMP_i \text{ is } LAST_i.] \\ &\Rightarrow SAFE_p \quad [SAFE_p = AV_p + ANT_p] \end{aligned}$$

Case(ii): $INSERT_{(i,j)}$.
$INSERT_{(i,j)}$ inserts the computation on edge (i, j), which is also shown to be safe.

$$\begin{aligned} INSERT_{(i,j)} &= \neg SPAVOUT_i.SPAVIN_j.SPANTIN_j \\ &\Rightarrow \neg SPAVOUT_i.SPAVIN_j \\ &\Rightarrow \neg AVIN_j.SPAVIN_j \quad [\neg SPAVOUT_i \Rightarrow \neg AVIN_j] \\ &\Rightarrow \neg AVIN_j.SAFEIN_j.PAVIN_j \quad [SPAVIN_j \Rightarrow SAFEIN_j.PAVIN_j] \\ &\Rightarrow \neg AVIN_j.SAFEIN_j \\ &\Rightarrow ANTIN_j \quad [SAFEIN_j = AVIN_j + ANTIN_j] \\ &\Rightarrow ANT_{(i,j)} \\ &\Rightarrow SAFE_{(i,j)} \end{aligned}$$

That is, all insertions of computations corresponding to the transformation are done at safe points. ∎

Lemma 2 *All candidate computations which are safe partially redundant become totally redundant after insertions corresponding to the transformation.*

Proof: We have to show that $SPREDUND_{i_f} \Rightarrow REDUND_{i_f}$, after insertions corresponding to the transformation.

From the definition of safe partial availability, we have,

$$\begin{aligned} & SPAVIN_i \\ & \Rightarrow \quad \exists p : p \in P[s,i] : [\exists m : m \in N \wedge m \in p : (COMP_m.TRANSP_{]m,i[}.SAFE_{[OUT_m,IN_i]})] \end{aligned}$$

where, s is the entry node and $P[s,i]$ is the set of all paths from node s to node i.

In order to select the earliest node m(from entry node s) on the path satisfying the required condition, we may write,

$$\begin{aligned} & SPAVIN_i \\ & \Rightarrow \quad \exists p : p \in P[s,i] : [\exists m : m \in N \wedge m \in p : (COMP_m.(\neg TRANSP_m + \neg SPAVIN_m) \\ & \qquad\qquad .TRANSP_{]m,i[}.SAFE_{[OUT_m,IN_i]})] \end{aligned}$$

Considering all paths from s to i, we may write,

$$\begin{aligned} & SPAVIN_i \\ & \Rightarrow \quad \forall p : p \in P[s,i] : [(\exists m : m \in N \wedge m \in p : (COMP_m.(\neg TRANSP_m + \neg SPAVIN_m) \\ & \qquad\qquad .TRANSP_{]m,i[}.SAFE_{[OUT_m,IN_i]})) \\ & \qquad + (\exists m,n : (m,n) \in E \wedge (m,n) \in p : ((\neg SPAVOUT_m.SPAVIN_n) \\ & \qquad\qquad .TRANSP_{[n,i[}.SAFE_{[IN_n,IN_i]}))] \end{aligned}$$

Refer to Fig.2(a) to see the two possibilities mentioned above, for any path.

Now, we have,

$$\begin{aligned} & SPREDUND_{i_f} = ANTLOC_i.SPAVIN_i \\ & \Rightarrow \quad \forall p : p \in P[s,i] : [(\exists m : m \in N \wedge m \in p : (COMP_m.(\neg TRANSP_m + \neg SPAVIN_m) \\ & \qquad\qquad .TRANSP_{]m,i[}.SAFE_{[OUT_m,IN_i]}.ANTLOC_i)) \\ & \qquad + (\exists m,n : (m,n) \in E \wedge (m,n) \in p : ((\neg SPAVOUT_m.SPAVIN_n) \\ & \qquad\qquad .TRANSP_{[n,i[}.SAFE_{[IN_n,IN_i]}.ANTLOC_i))] \\ & \Rightarrow \quad \forall p : p \in P[s,i] : [(\exists m : m \in N \wedge m \in p : (COMP_m.(\neg TRANSP_m + \neg SPAVIN_m) \\ & \qquad\qquad .SPANTOUT_m)) \\ & \qquad + (\exists m,n : (m,n) \in E \wedge (m,n) \in p : (\neg SPAVOUT_m.SPAVIN_n \\ & \qquad\qquad .SPANTIN_n))] \end{aligned}$$

[From, $ANTLOC_i.TRANSP_{]m,i[}.SAFE_{[OUT_m,IN_i]} \Rightarrow SPANTOUT_m$ and $ANTLOC_i.TRANSP_{[n,i[}.SAFE_{[IN_n,IN_i]} \Rightarrow SPANTIN_n$]

Rearranging the terms we get,

$$\begin{aligned} & SPREDUND_{i_f} \\ & \Rightarrow \quad \forall p : p \in P[s,i] : [(\exists m : m \in N \wedge m \in p : (COMP_m.SPANTOUT_m \\ & \qquad\qquad .(\neg TRANSP_m + \neg SPAVIN_m))) \\ & \qquad + (\exists m,n : (m,n) \in E \wedge (m,n) \in p : (\neg SPAVOUT_m.SPAVIN_n \\ & \qquad\qquad .SPANTIN_n))] \\ & \Rightarrow \quad \forall p : p \in P[s,i] : [(\exists m : m \in N \wedge m \in p : INSERT_m) \\ & \qquad + (\exists m,n : (m,n) \in E \wedge (m,n) \in p : INSERT_{(m,n)})] \end{aligned}$$

[From, $INSERT_m = COMP_m.SPANTOUT_m.(\neg TRANSP_m + \neg SPAVIN_m)$

and $INSERT_{(m,n)} = \neg SPAVOUT_m.SPAVIN_n.SPANTIN_n]$

$\Rightarrow$ $AVIN_i$, after insertions ($INSERT_m$ or $INSERT_{(m,n)}$), on all paths.

$\Rightarrow$ $REDUND_{i_f}$ $[ANTLOC_i.AVIN_i \Rightarrow REDUND_{i_f}]$

That is, all safe partially redundant computations in the original program become totally redundant after insertions corresponding to the transformation. ∎

Lemma 3 *Only those candidate computations which would be redundant after insertions corresponding to the transformation are replaced.*

Proof: We have to show that after insertions corresponding to the transformation

$REPLACE_{i_f} \Rightarrow REDUND_{i_f}$, and
$REPLACE_{i_l} \Rightarrow REDUND_{i_l}$

First, let us show that $REPLACE_{i_l} \Rightarrow REDUND_{i_l}$. We have,

$$\begin{aligned} REPLACE_{i_l} &= COMP_i.(SPANTOUT_i + TRANSP_i.SPAVIN_i) \\ &= COMP_i.SPANTOUT_i + COMP_i.TRANSP_i.SPAVIN_i \end{aligned}$$

Let us consider it as two separate cases:
Case(i) : $REPLACE_{i_l} = COMP_i.SPANTOUT_i$

$$\begin{aligned} REPLACE_{i_l} &= COMP_i.SPANTOUT_i \\ &= COMP_i.SPANTOUT_i.(SPAVIN_p + \neg SPAVIN_p), \\ &\quad \text{where } p \text{ is the point just before } LAST_i. \\ &= COMP_i.SPANTOUT_i.SPAVIN_p \\ &\quad + COMP_i.SPANTOUT_i.\neg SPAVIN_p \end{aligned}$$

Let us consider the above equation also as two separate cases:
Case(i)a : $REPLACE_{i_l} = COMP_i.SPANTOUT_i.SPAVIN_p$

$$\begin{aligned} REPLACE_{i_l} &= COMP_i.SPANTOUT_i.SPAVIN_p \\ &\Rightarrow COMP_i.SPAVIN_p \\ &\Rightarrow COMP_i.SPAVIN_p.TRANSP_i \ [COMP_i.SPAVIN_p \Rightarrow TRANSP_i] \\ &\Rightarrow ANTLOC_i.SPAVIN_i \\ &\quad [COMP_i.TRANSP_i \Rightarrow ANTLOC_i; \\ &\quad\ SPAVIN_p.TRANSP_i \Rightarrow SPAVIN_i] \\ &\Rightarrow SPREDUND_{i_f} \quad [SPREDUND_{i_f} = ANTLOC_i.SPAVIN_i] \\ &\Rightarrow REDUND_{i_f} \quad [\text{From Lemma 2}] \\ &\Rightarrow REDUND_{i_l} \quad [COMP_i.TRANSP_i \Rightarrow FIRST_i = LAST_i] \end{aligned}$$

Case(i)b : $REPLACE_{i_l} = COMP_i.SPANTOUT_i.\neg SPAVIN_p$

$$\begin{aligned} REPLACE_{i_l} &= COMP_i.SPANTOUT_i.\neg SPAVIN_p \\ &\Rightarrow INSERT_i \quad [\text{From the definition of } INSERT_i] \\ &\Rightarrow REDUND_{i_l} \quad [COMP_i.INSERT_i \Rightarrow REDUND_{i_l}] \end{aligned}$$

Case(ii) : $REPLACE_{i_l} = COMP_i.TRANSP_i.SPAVIN_i$

$$\begin{aligned} REPLACE_{i_l} &= COMP_i.TRANSP_i.SPAVIN_i \\ &\Rightarrow ANTLOC_i.SPAVIN_i \quad [COMP_i.TRANSP_i \Rightarrow ANTLOC_i] \\ &\Rightarrow SPREDUND_{i_f} \quad [SPREDUND_{i_f} = ANTLOC_i.SPAVIN_i] \\ &\Rightarrow REDUND_{i_f} \quad \text{[From Lemma 2]} \\ &\Rightarrow REDUND_{i_l} \quad [COMP_i.TRANSP_i \Rightarrow FIRST_i = LAST_i] \end{aligned}$$

Now, let us show that $REPLACE_{i_f} \Rightarrow REDUND_{i_f}$, after insertions corresponding to the transformation. We have,

$$\begin{aligned} REPLACE_{i_f} &= ANTLOC_i.(SPAVIN_i + TRANSP_i.SPANTOUT_i) \\ &= ANTLOC_i.SPAVIN_i + ANTLOCi.TRANSP_i.SPANTOUT_i \end{aligned}$$

Let us consider it as two separate cases
Case(i) : $REPLACE_i = ANTLOC_i.SPAVIN_i$

$$\begin{aligned} REPLACE_{i_f} &= ANTLOC_i.SPAVIN_i \\ &\Rightarrow SPREDUND_{i_f} \quad [SPREDUND_{i_f} = ANTLOC_i.SPAVIN_i] \\ &\Rightarrow REDUND_{i_f} \quad \text{[From Lemma 2]} \end{aligned}$$

Case(ii) : $REPLACE_{i_f} = ANTLOC_i.TRANSP_i.SPANTOUT_i$

$$\begin{aligned} REPLACE_{i_f} &= ANTLOC_i.TRANSP_i.SPANTOUT_i \\ &\Rightarrow COMP_i.SPANTOUT_i \quad [ANTLOC_i.TRANSP_i \Rightarrow COMP_i] \\ &\Rightarrow REDUND_{i_l} \quad \text{[From Case(i) of } REPLACE_{i_l} \text{ above]} \\ &\Rightarrow REDUND_{i_f} \quad [ANTLOC_i.TRANSP_i \Rightarrow FIRST_i = LAST_i] \end{aligned}$$

We have shown that all candidate computations which are replaced are the ones which become redundant after insertions corresponding to the transformation. ∎

Lemma 4 *After the transformation no path contains more computations of an expression than it contained before.*

Proof: We have to show that there is at least one replacement corresponding to each insertion on a path.
Let us consider the two cases of insertions:
Case(i) : $INSERT_i$

$$\begin{aligned} INSERT_i &= COMP_i.SPANTOUT_i.(\neg TRANSP_i + \neg SPAVIN_i) \\ &\Rightarrow COMP_i.SPANTOUT_i \\ &\Rightarrow REPLACE_{i_l} \quad \text{[From the definition of } REPLACE_{i_l}] \end{aligned}$$

i.e., Any path involving node i, corresponding to $INSERT_i$, has at least one replacement, $REPLACE_{i_l}$.
Case(ii) : $INSERT_{(i,j)}$

$$\begin{aligned} INSERT_{(i,j)} &= \neg SPAVOUT_i.SPAVIN_j.SPANTIN_j \\ &\Rightarrow ANTIN_j \quad \text{[From Case(ii), Lemma 1]} \\ &\Rightarrow \forall p : p \in P[j,e] : (\exists k : SAFE_{[IN_j,IN_k]}.TRANSP_{[j,k[}.ANTLOC_k) \\ &\quad \text{where, } e \text{ is the exit node.} \end{aligned}$$

$$
\begin{aligned}
\Rightarrow \quad & \forall p : p \in P[j,e] : (\exists k : ANTLOC_k.SPAVIN_k) \\
& [SAFE_{[IN_j,IN_k]}.TRANSP_{[j,k[}.SPAVIN_j \Rightarrow SPAVIN_k] \\
\Rightarrow \quad & \forall p : p \in P[j,e] : (\exists k : REPLACE_{k_f}) \\
& [ANTLOC_k.SPAVIN_k \Rightarrow REPLACE_{k_f}]
\end{aligned}
$$

i.e., Any path involving edge (i, j), corresponding to $INSERT_{(i,j)}$, has at least one replacement, $REPLACE_{k_f}$.
From case(i) and case(ii), we conclude that any insertion on a path implies at least one replacement on the same path. ∎

Theorem 1 *The algorithm performs partial redundancy elimination correctly.*

Proof: By Lemmas 1, 2, 3, and 4. ∎

5.2 Computational Optimality

Lemma 5 *A candidate computation is not replaced by the transformation if and only if it is an isolated computation.*

Proof: We have to show that

$$
\begin{aligned}
ANTLOC_i.\neg REPLACE_{i_f} &= ISOLATED_{i_f}, \text{ and} \\
COMP_i.\neg REPLACE_{i_l} &= ISOLATED_{i_l}
\end{aligned}
$$

Case(i) : $ANTLOC_i.\neg REPLACE_{i_f} = ISOLATED_{i_f}$

$$
\begin{aligned}
REPLACE_{i_f} &= ANTLOC_i.(SPAVIN_i + TRANSP_i.SPANTOUT_i) \\
\neg REPLACE_{i_f} &= \neg ANTLOC_i + \neg(SPAVIN_i + TRANSP_i.SPANTOUT_i)
\end{aligned}
$$

Hence,

$$
\begin{aligned}
& ANTLOC_i.\neg REPLACE_{i_f} \\
&= ANTLOC_i.\neg(SPAVIN_i + TRANSP_i.SPANTOUT_i) \\
&= ANTLOC_i.\neg SPAVIN_i.\neg(TRANSP_i.SPANTOUT_i) \\
&= ISOLATED_{i_f}
\end{aligned}
$$

Case(ii) : $COMP_i.\neg REPLACE_{i_l} = ISOLATED_{i_l}$

$$
\begin{aligned}
REPLACE_{i_l} &= COMP_i.(SPANTOUT_i + TRANSP_i.SPAVIN_i) \\
\neg REPLACE_{i_l} &= \neg COMP_i + \neg(SPANTOUT_i + TRANSP_i.SPAVINi)
\end{aligned}
$$

Hence,

$$
\begin{aligned}
& COMP_i.\neg REPLACE_{i_l} \\
&= COMP_i.\neg(SPANTOUT_i + TRANSP_i.SPAVIN_i) \\
&= COMP_i.\neg SPANTOUT_i.\neg(TRANSP_i.SPAVIN_i) \\
&= ISOLATED_{i_l}
\end{aligned}
$$

i.e., All candidate computations which are not replaced by the transformation are isolated computations and vice versa. ∎

Theorem 2 *The transformation is computationally optimal.*

Proof: We have to show that there does not exist any other correct transformation with less number of computations of an expression on any path.

Let us assume that there exists another correct transformation with less number of computations of an expression on a path. This is possible only under two situations:
Case(i) : Number of replacements on the path is more in the new transformation, without a corresponding increase in insertions. This

$\Rightarrow$ a computation which was not replaced by our transformation is replaced by the new one without an additional insertion.
$\Rightarrow$ the new transformation replaces an isolated computation without a corresponding insertion. [By Lemma 5]
$\Rightarrow$ incorrect transformation

Case(ii) : Number of insertions on the path is less in the new transformation.
This implies that some of the insertions in our transformation were unnecessary. Let us consider the two cases of insertions:

Case (ii)a : $INSERT_i$
Let us assume that the insertion corresponding to $INSERT_i$ in our transformation was unnecessary and hence not done.

$INSERT_i \Rightarrow REPLACE_{i_l}$ [From case(i), Lemma 4]
$\Rightarrow$ the new variable introduced by $REPLACE_{i_l}$ in node i has no initialization. [From the assumption, case(ii)a]
$\Rightarrow$ incorrect transformation

Case(ii)b : $INSERT_{(i,j)}$
Let us assume that the insertion corresponding to $INSERT_{(i,j)}$ in our transformation was unnecessary and hence not done.

$INSERT_{(i,j)} \Rightarrow \forall p : p \in P[j,e] : (\exists k : REPLACE_{k_f})$ [From case(ii), Lemma 4]
$\Rightarrow$ the new variable introduced by $REPLACE_{k_f}$ in node k has no initialization along the path involving edge (i,j). [From the assumption, case(ii)b]
$\Rightarrow$ incorrect transformation

Both case(i) and case(ii) lead to incorrect transformation, contradicting our assumption that there exists another correct transformation with less number of computations of an expression on a path.
Hence, we conclude that the transformation is computationally optimal. ∎

5.3 Lifetime Optimality

Theorem 3 *The transformation is lifetime optimal.*

Proof: We have to show that the transformation keeps the live ranges of the newly introduced temporaries to the minimum.
Let us assume that the live ranges of the temporaries introduced by our transformation is not minimal. This implies that there exists at least one insertion which can be moved to a later point in the flow graph, preserving correctness of the algorithm. Let us consider the two cases

of insertions:

Case(i) : $INSERT_i$.
Let us insert the computation just after $LAST_i$, the next later point in the flow graph, instead of just before it. This

$\Rightarrow$ the new variable introduced by $REPLACE_{i_l}$ in node i has no initialization
[$INSERT_i \Rightarrow REPLACE_{i_l}$. From case(i), Lemma 4]

$\Rightarrow$ incorrect transformation

Similarly, insertion at any other later points, instead of insertion just before $LAST_i$, also leads to incorrect transformation.

Case(ii) : $INSERT_{(i,j)}$.
Let us insert the computation at the entry of node j, the next later point in the flow graph, instead of insertion on the edge (i, j). This

$\Rightarrow$ number of computations on paths involving other edges to j
is increased by one. Note that $\mid Pred(j) \mid > 1$.

$\Rightarrow$ the transformation is not computationally optimal

Similarly, insertion at any other later point also leads to violation of computational optimality. In both case(i) and case(ii) insertion at a later point leads to the violation of either the correctness or the computational optimality of the transformation, contrary to our assumption.
Hence, the transformation is lifetime optimal. ∎

6 An Example

Consider the following example - same as in [11] - in Figure 3.

Local Properties:

$$\begin{aligned} ANTLOC &= \{2,4,7,8,9\} \\ COMP &= \{2,4,7,9\} \\ TRANSP &= \{1,3,4,5,6,7,9,10\} \end{aligned}$$

Global Boolean Properties:

$$\begin{aligned} AVIN &= \phi \\ AVOUT &= \{2,4,7,9\} \\ ANTIN &= \{2,4,5,6,7,8,9\} \\ ANTOUT &= \{4,5,6,7,8\} \\ SAFEIN &= \{2,4,5,6,7,8,9\} \\ SAFEOUT &= \{2,4,5,6,7,8,9\} \\ SPAVIN &= \{4,5,8\} \\ SPAVOUT &= \{2,4,5,7,9\} \\ SPANTIN &= \{2,4,5,6,7,8,9\} \\ SPANTOUT &= \{4,5,6,7,8\} \end{aligned}$$

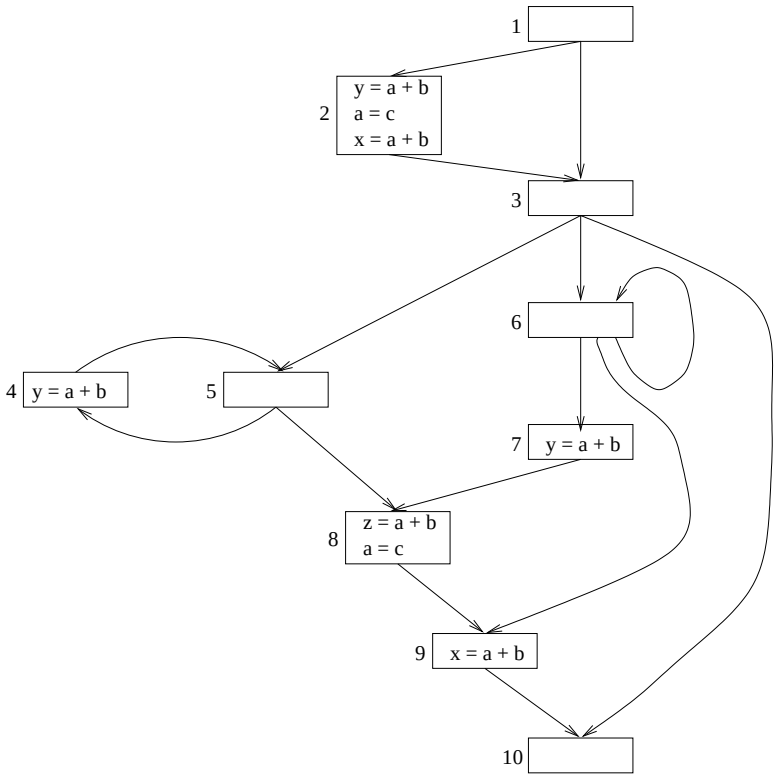

Figure 3: Initial program

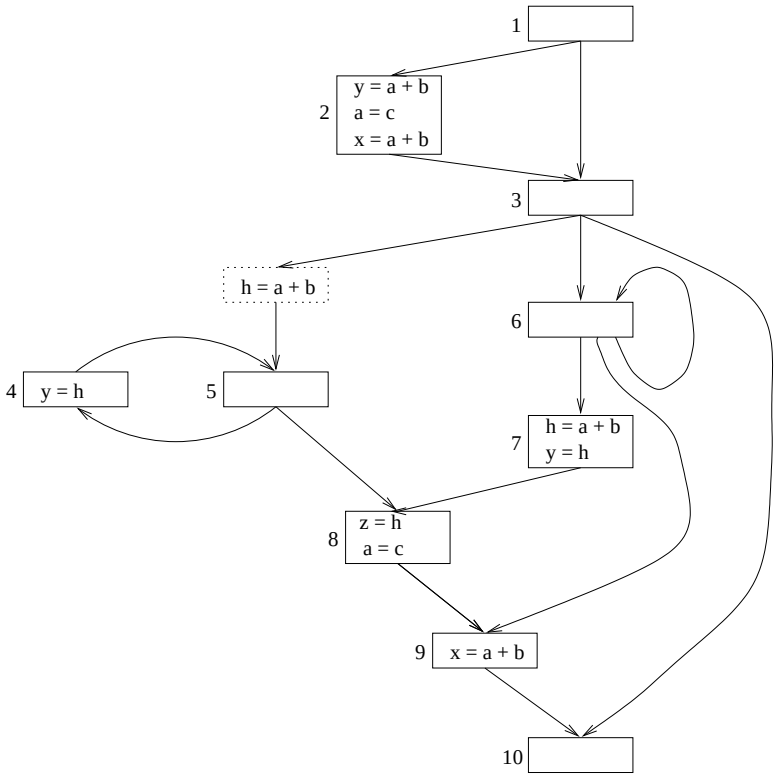

Figure 4: Transformed program

Insertions and Replacements:

$$\begin{aligned} INSERT_i &= \{7\} \\ INSERT_{(i,j)} &= \{(3,5)\} \\ REPLACE_{i_f} &= \{4,7,8\} \\ REPLACE_{i_l} &= \{4,7\} \end{aligned}$$

The solution is insertions just before the *last computation* in node 7 and on edge (3, 5), and replacement of the *first computation* in nodes 4, 7, and 8 and replacement of the *last computations* in nodes 4 and 7. The transformed program is given in Figure 4.

7 Conclusion

We have presented a simple, optimal, and practical algorithm for partial redundancy elimination. The algorithm is conceptually simple as it evolves naturally from the new concept of safe partial availability. The proof of correctness of the algorithm also becomes simple with the new concept of safe partial availability as the basis of the argument. The algorithm is computationally and lifetime optimal. It works on flow graphs whose nodes are basic blocks which is the standard in optimizing compilers. In comparison with its predecessor [11], our algorithm also requires four unidirectional analyzes but it does not require the edge splitting transformations to be done before its application.

References

[1] A.V.Aho, R.Sethi, and J.D.Ullman, Compilers: Principles, Techniques, and Tools (Addison-Wesley, 1986).

[2] F.Chow, A portable machine independent optimizer - Design and implementation, PhD Thesis, Dept. of Electrical Engineering, Stanford University, Stanford, Calif., and Tech. Rep. 83-254, Computer Systems Lab., Stanford University, 1983.

[3] V.M.Dhaneshwar and D.M.Dhamdhere, Strength reduction of large expressions, Journal of Programming Languages 3 (1995) 95-120.

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

[5] D.M.Dhamdhere, Practical adaptation of the global optimization algorithm of Morel and Renvoise, ACM TOPLAS 13, 2(1991) 291-294.

[6] D.M.Dhamdhere and U.P.Khedker, Complexity of bidirectional data-flow analysis, in: Conference record of 20th ACM Symposium on the Principles of Programming Languages (ACM, New York, 1993) 397-409.

[7] D.M.Dhamdhere and H.Patil, An elimination algorithm for bidirectional data-flow problems using edge placement, ACM TOPLAS 15, 2(1993) 312-336.

[8] D.M.Dhamdhere, B.K.Rosen, and F.K.Zadeck, How to analyze large programs efficiently and informatively, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation '92, ACM SIGPLAN Notices 27, 7(1992) 212-223.

[9] K.H.Dreshler and M.P.Stadel, A variation of Knoop, Rüthing, and Steffen's lazy code motion, ACM SIGPLAN Notices 28, 5(1993) 29-38.

[10] J.Knoop, O.Rüthing, and B.Steffen, Lazy code motion, ACM SIGPLAN Notices 27, 7(1992) 224-234.

[11] J.Knoop, O.Rüthing, and B.Steffen, Optimal code motion: theory and practice, ACM TOPLAS 16, 4(1994) 1117-1155.

[12] E.Morel and C.Renvoise, Global optimization by suppression of partial redundancies, Comm. of the ACM 22, 2(1979) 96-103.

[13] V.K.Paleri, Y.N.Srikant, and P.Shankar, A simple algorithm for partial redundancy elimination, ACM SIGPLAN Notices 33, 12(1998) 35-43.