STRENGTH REDUCTION OF LARGE EXPRESSIONS

Vikram M. Dhaneshwar* Dhananjay M. Dhamdhere†

Department of Computer Science and Engineering

Indian Institute of Technology, Bombay 400 076 (India).

Journal of Programming Languages, **3** (1995), 95-120.

Abstract

Strength reduction of a large expression cannot be performed as a second order effect of optimizing its subexpression(s). We use the ideas of composite hoisting and strength reduction [1, 2, 3] to develop an algorithm for the strength reduction of a large expression (SRLE) as a single entity using the framework for partial redundancy elimination [4, 5, 6]. SRLE subsumes the conventional optimizations of code hoisting, common subexpression elimination, loop invariant movement and strength reduction, thereby performing comprehensive optimization of a program. It performs partial redundancy elimination as effectively as the recent algorithms [5, 21, 22], and also eliminates several suboptimalities of previously published work in partial-redundancy–based strength reduction [2, 3].

The SRLE algorithm involves use of nonsingular bidirectional data flows, giving rise to interesting issues concerning the desired fixed point of the data flows. A solution method based on the principle of decomposition of bidirectional data flows into a sequence of unidirectional data flows is shown to achieve the desired fixed point of the bidirectional data flows more elegantly than earlier methods. Experimental results as well as proof of correctness of the algorithm are included.

Keywords :

Code optimization, Strength reduction, Dead code elimination, Large expressions, Data flow analysis, Decomposition of data flows, Fixed point.

*present address : Tata Research Centre, Pune 411 050, email : vmd@surya.trddc.ernet.in

†dmd@cse.iitb.ernet.in

1 Introduction

Unifications of several classical optimizing transformations within one optimization framework have been motivated by the desire to reduce the cost of optimization [4, 7, 2, 3]. The Morel-Renvoise algorithm for optimization by elimination of partial redundancies performs code movement, common subexpression elimination and loop invariant movement within a unified optimization framework [4]. Joshi and Dhamdhere [2] incorporated the work of Dhamdhere [1] and Dhamdhere and Isaac [7] within the partial redundancy elimination framework of Morel-Renvoise. They called their algorithm the composite hoisting and strength reduction algorithm (CHSA) [2]. This algorithm enhances the scope of optimization by realizing optimizations not achievable by mere compositions of the classical optimizations it subsumes. Successive algorithms for composite hoisting and strength reduction have attempted to improve the practicability and profitability of optimization [7, 2, 3]. However these algorithms share one limitation with the early works on strength reduction [8, 9] — they restrict their scope to *elementary* candidate expressions, that is, expressions involving a single operator.

Strength reduction of a large expression[1] cannot be performed as a second order effect of optimizing its subexpression(s). For example, no second order optimization is possible for the expression $i*5+j*10$ following the strength reduction of the elementary subexpressions $i*5$ and $j*10$ whereas strength reduction of $i*5+j*10$ as a single entity can produce a superior result due to more dead code elimination (see section 2 for an illustration). Thus, a different approach is required for effective strength reduction of large expressions.

In this paper, we develop an algorithm for strength reduction of *large* candidate expressions (SRLE). The algorithm eliminates many suboptimalities inherent in earlier work on strength reduction, and achieves substantial improvements in the overall profitability of optimization in practice (see section 6).

Two important issues concerning the solution of data flows that are used in the unified optimization framework are also addressed in this paper, namely efficient solution methods and desired fixed points of the data flows. The data flows used in unified optimization frameworks are *nonsingular bidirectional* in nature [5, 10, 11], that is, they contain multiple confluence operators — in this case, both the $\sum$ and $\prod$ operators. The classical techniques of efficient data flow analysis are not applicable to these flows. We present a solution approach based on the principle (developed in [10, 11]) of decomposition of bidirectional data flows into a sequence of unidirectional data flows. This approach

[1]For the purpose of this paper, a *large* expression is any expression involving two or more operators.

makes the classical techniques of data flow analysis applicable to the unified optimization frameworks as well.

The latter issue concerns the desired solution of the code placement problem underlying the unified algorithm. It has been shown [12] that the desired fixed point of a nonsingular bidirectional data flow is different from the maximal fixed point according to any of the confluences. While Dhamdhere, Patil [12] provide a method to obtain the desired fixed point, their method is complex to program. In this paper we propose a simple and elegant solution method for the decomposed data flows that provides the desired fixed point.

This paper assumes familiarity with the basic concepts used in control and data flow analysis [13]. Section 2.2 introduces the composite hoisting and strength reduction optimization. Section 2.3 defines the notion of large expressions and presents an example of the strength reduction of large expressions. The desired placement of computations during composite hoisting and strength reduction is described in section 3. Section 3.2 describes the code placement policy used by the proposed algorithm. Some important properties of the placement are proved in section 3.3. Section 4 presents the Strength Reduction algorithm for Large Expressions (SRLE) and its data flow equations. Section 4.1 describes the solution method for its data flows. Section 5 presents a comparison with earlier work. Section 6 presents experimental results comparing the effectiveness of the optimization and the solution efficiency of the SRLE algorithm with the NCHSA algorithm proposed previously [3]. The correctness of the SRLE algorithm is proved in the appendix.

2 Preliminaries

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 [13].

2.1 Simple hoisting

Anticipatability and Safety

An expression e is *locally available* in node i if 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 i if 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 anticipatable or available at that point [14]. 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. In practice, safety has often been treated to mean anticipatability alone because it is unnecessary to place a computation of e at a point where it is available. However, using the full scope of the definition of safety facilitates our presentation of the code placement criteria.

Simple hoisting (also called *partial redundancy elimination*) involves placement of computations of e at suitable program points so as to make some partially redundant occurrences of e wholly redundant.

2.2 Composite hoisting and strength reduction optimization

The transformation of *strength reduction* is the replacement of an application of a high strength operator α by repeated applications of a low strength operator β [15]. The rationale for the replacement is that a low strength operator consumes less execution time than a high strength operator. Typical examples of the (α, β) pairs are (multiplication, addition) and (exponentiation, multiplication). In this paper we restrict ourselves to the (multiplication, addition) pair. *Composite hoisting and strength reduction* combines optimization by simple code hoisting with optimization by strength reduction.

Update definition and Redefinition of a variable

A definition $v :=< expression >$ is an *update definition* of v if $< expression >$ is of the form $v := v \pm c$, where c is a constant. A *redefinition* is any definition other than an update definition.

Update computation and Recomputation of an expression

An *expression variable* is a variable T_e introduced in the optimized program to hold the value of an expression e subjected to optimization. An *update computation* for an expression e updates the current value of T_e by an appropriately chosen constant g; it has the form $T_e := T_e + g$. A *recomputation* for e assigns the current value of e to T_e; it has the form $T_e := e$.

Consider all paths $[i \ldots k]$[2] such that node k contains an occurrence of expression $e = v * c$, where v is an integer variable and c is an integer constant, each path $(i \ldots k)$ is free of occurrences of e or definitions of v, and either $i = n_0$, the program entry node, or node i contains a definition of v. *Composite hoisting and strength reduction* of expression e occurring in node k can be performed by replacing the use of expression e by the use of its expression variable T_e after performing the following code placements[3] [7, 2, 3]:

(a) An update computation of e is placed in node i if i contains an update definition of an operand of e and e is available at the entry to node i,

(b) If condition (a) is not satisfied, a recomputation of e is placed in node i.

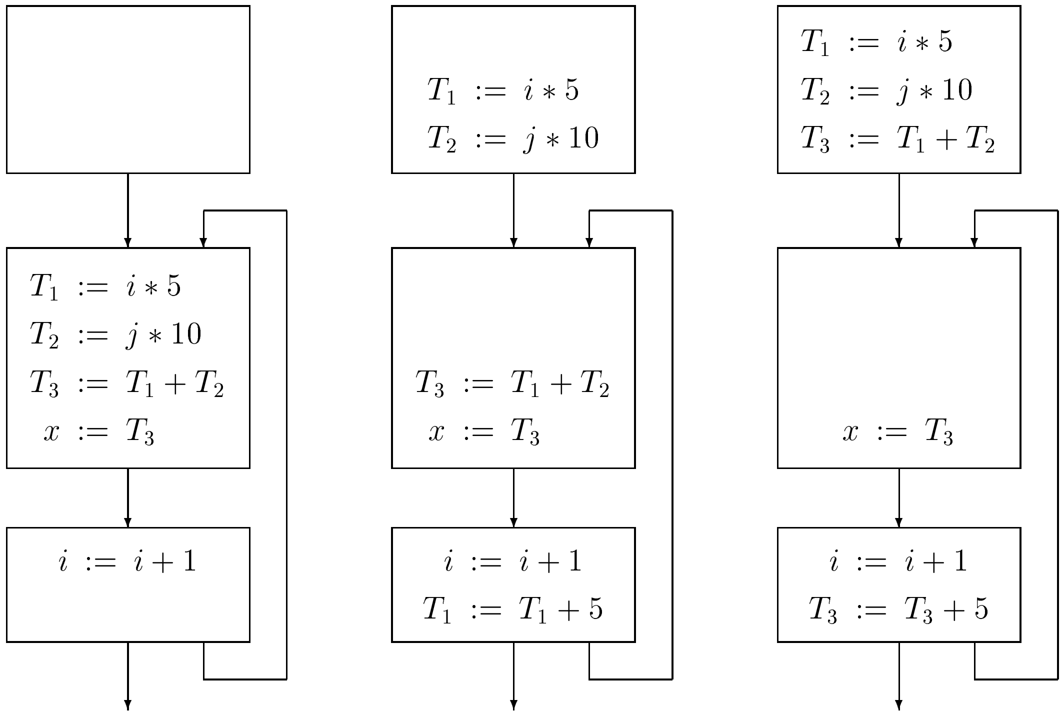
Note that the above rules perform strength reduction for e only if v is subjected to update definitions, else hoisting is performed. Further, hoisting as performed here subsumes common subexpression elimination and loop optimization as described in [4, 7], hence an application of the composite hoisting and strength reduction algorithm results in comprehensive optimization of a program.

2.3 Composite hoisting and strength reduction of large expressions

Many traditional optimizers restrict their scope to elementary expressions, that is expressions involving a single operator, and optimize large expressions by repeated applications of the optimization (see [16] for an example). However, this approach is not effective for large expressions involving high strength subexpressions. For example, for the program of figure 1(a) conventional optimization would separately perform strength reduction of $i*5$ and hoisting of $j*10$ (see figure 1(b)), whereas strength reduction of $i*5+j*10$ as a single entity would have achieved more dead code elimination (see figure 1(c)). Joshi [17] and Drechsler and Stadel [18] discuss hoisting of large expressions within the partial redundancy elimination framework of Morel and Renvoise [4]. We extend the scope of optimization of large expressions to composite hoisting and strength reduction along the

[2]We denote a path from a node i to a node k, both inclusive, by $[i \ldots k]$. A path from i to k, including node i but not node k, is denoted by $[i \ldots k)$. $(i \ldots k)$ and $(i \ldots k]$ are analogously defined.

[3]Such optimization is performed subject to *profitability* and *safety* criteria, which we discuss in section 3.



(a) (b) (c)

Figure 1: (a) 3-address form of a program containing the large expression $i*5+j*10$, (b) Conventional optimization, (c) Large expression optimization (*note :* expression variables T_1, T_2 and T_3 are used instead of T_{i*5}, T_{j*10} and $T_{i*5+j*10}$ in the interest of simplicity)

lines proposed in Dhamdhere [3] and Joshi and Dhamdhere [2]. This leads to higher profitabilities due to increased dead code elimination.

Definition 2.1 *A* candidate expression *for composite hoisting and strength reduction optimization of large expressions is an integer expression in one of the following forms :*

(i) $v * c$ *or* $c * v$,

(ii) $e_1 \pm k$ *or* $k \pm e_1$,

(iii) $e_1 \pm e_2$,

(iv) $e_1 * c$ *or* $c * e_1$,

where v *is a variable,* c *is a constant,* k *is a variable or a constant and* e_1, e_2 *are themselves candidate expressions.*

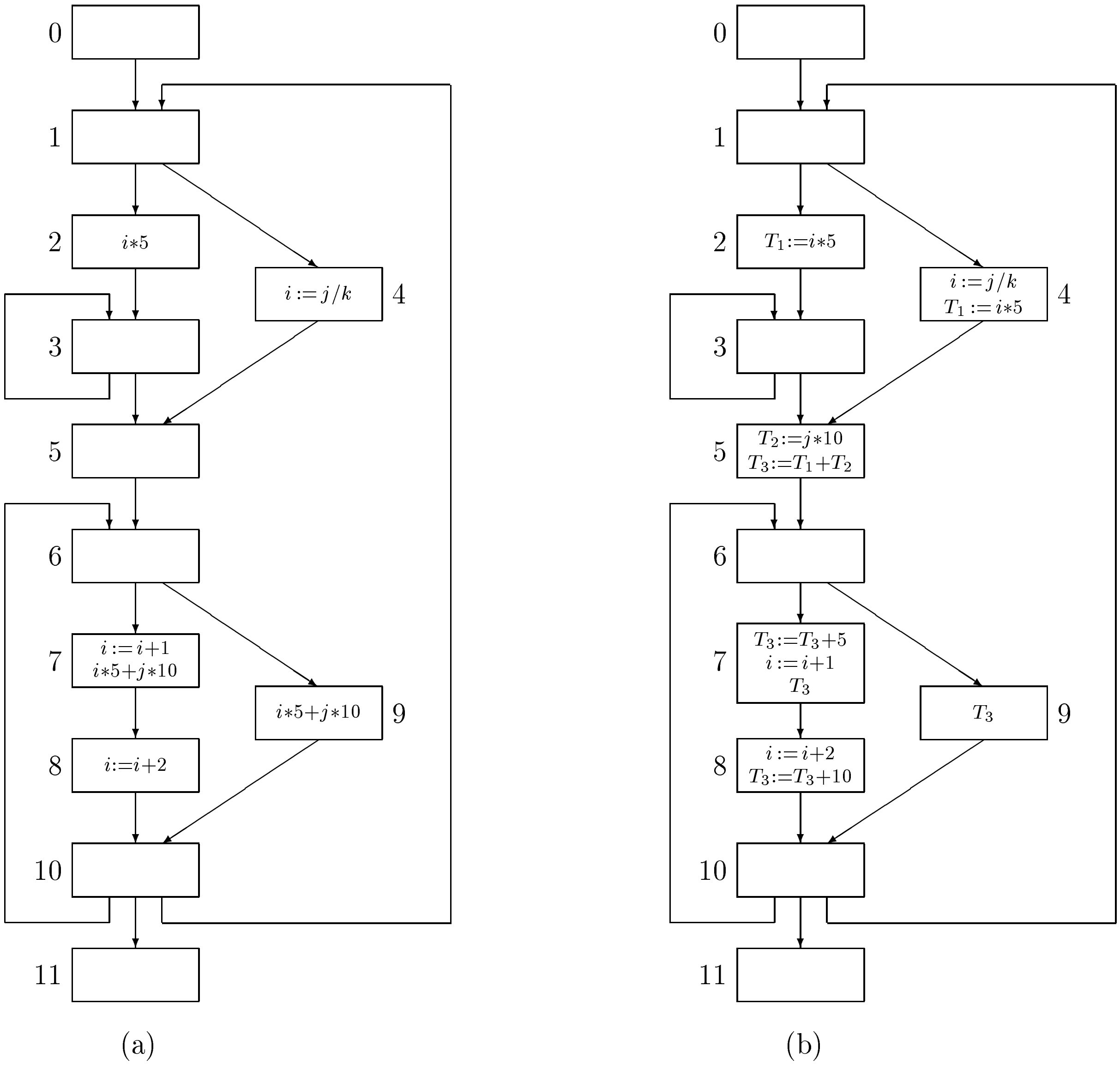


Figure 2: Composite hoisting and strength reduction of large expressions (*note :* expression variables T_1, T_2 and T_3 are used instead of T_{i*5}, T_{j*10} and $T_{i*5+j*10}$ in the interest of simplicity)

Figure 2 illustrates optimization of a large expression $i*5+j*10$ using our approach to the composite hoisting and strength reduction optimization of large expressions. Note that the elementary expressions $i*5$ and $j*10$ as well as the large expression $i*5+j*10$ are subjected to composite hoisting and strength reduction in the loop {6,7,8,9,10}. Expression $j*10$ is simply hoisted out of the loop, while expressions $i*5$ and $i*5+j*10$ are strength reduced. Nodes 7 and 8 contain update computations for the expression $i*5+j*10$. Update computations for $i*5$ would also have been introduced in nodes 7 and 8 (these would have the form $T_1 := T_1+5$ and $T_1 := T_1+10$, respectively). However, these are not shown here because they would be removed during dead code elimination.

3 Desired Placement of Computations

In this section, we consider the placement of inserted computations during composite hoisting and strength reduction. We first describe the placement of expressions subject to simple hoisting, and then extend the placement to include expressions subject to strength reduction.

3.1 Placement of computations in simple hoisting

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

Definition 3.1 *An occurrence of an expression e in a node k is an* eliminatable occurrence of e *if the occurrence is locally anticipatable in k and there exists a path $[i \ldots k]$ such that :*

(a) e is locally available at the exit of i,

(b) All nodes on the path $(i \ldots k)$ are empty with respect to e, and

(c) e is safe at the exit of each node on the path $[i \ldots k)$.

An expression e is *eliminatable from a node k* iff there is an eliminatable occurrence of e in node k or a computation of e placed at the exit of k would be an eliminatable occurrence of e. In figure 2(a), the expression $j*10$ is eliminatable from node 7 of the flow graph due to the presence of the paths $7-8-10-6-7$ and $9-10-6-7$.

Let node k contain an eliminatable occurrence of an expression e and let node i contain a computation of e as per definition 3.1. Let S be the set of nodes on the path $(i \ldots k]$.

The occurrence of e in node k can be made redundant by inserting a computation of e at the exit of all nodes l such that

(i) l is a predecessor of some node $m \in S$,

(ii) $l \notin S$, and

(iii) e is safe, but not available, at the exit of l.

If the computation is not safe at the exit of l then it can be placed in a synthetic node lm inserted along the edge (l, m).

The safety requirement ensures that no new expressions are inserted along any path in the program, however the execution time of a program may increase due to such placement. For example, consider a path $l \ldots n$ such that node n contains an occurrence of e and e is safe at the exit of l. Now, insertion of e at the exit of l would degrade the execution efficiency of the program along path $l \ldots n$ unless the occurrence of e can be eliminated from n. Hence, for profitable optimization we must ensure that all paths starting at the exit of l have a prefix $[l \ldots k]$ such that k contains an eliminatable occurrence of e. A similar requirement exists if the computation is placed in the synthetic node lm. Further, e must not be eliminatable in node l (else that would be a partial redundancy we would remove subsequently). Thus we arrive at the following criteria for the placement of hoisted computations.

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

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

a. e is not available at the exit of node l,

b. e is not eliminatable from node l, and

c. All paths starting at the exit of l have a prefix $[l \ldots k]$ such that k contains an eliminatable occurrence of e, and for all nodes j on the path $(l \ldots k)$, e is eliminatable from j and j is empty with respect to e.

[DPH–2] A computation of e is placed in a synthetic node lm on the edge (l, m) iff

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

b. e is not available at the exit of node l, and

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

In figure 2(a) node 5 satisfies the criterion DPH–1 for the expression $j * 10$.

Early algorithms for hoisting and strength reduction suffer from the problem of *redundant hoisting*, that is hoisting not accompanied by execution profits (see [5] for an example). By DPH–1 and DPH–2, a necessary condition for an expression hoisted out of node k to be inserted into a node l (or node lm) is that it is eliminatable from each of its descendants along the path $(l \ldots k]$, hence it follows that an expression is hoisted only as far as necessary to eliminate an eliminatable partial redundancy and no farther. A proof of this fact is given in section 3.3.

In this paper, we assume that the computation of a large expression e is inserted *after* the computations of its subexpressions (if any) inserted into the same node. This ensures the availability of the constituent subexpressions at the point of computation of e.

3.2 Placement of computations in composite hoisting and strength reduction

A simple way to extend the above optimization to strength reduction is the approach taken by Joshi and Dhamdhere [2], which ignores the presence of update definitions while defining availability, anticipatability and empty nodes (they called these the *X-properties*, for example *X-anticipatability* of an expression, *X-empty* nodes and so on). Recomputations of an expression e are then inserted using the algorithm for simple hoisting. The expression variable for e is suitably updated by inserting an update computation of e after every update definition of its operands along every path from an original or inserted recomputation of e to an eliminated occurrence of e. This simple approach suffers from certain difficulties, for example the execution time of a path may increase due to the placement of a recomputation followed by one or more update computations along a path, or placement of many update computations along a path and so on. Much effort is directed in [2, 3] towards the elimination of such suboptimalities.

We follow an analogous approach to perform composite hoisting and strength reduction, with the following difference : To eliminate the effort involved in inhibiting the placement of many update computations along a path, we confine our attention to the set of paths along which only a bounded number of update definitions occur. We begin by introducing two categories of data flow properties — the ones used for hoisting purposes take the prefix *hoist*, while the ones used for strength reduction purposes take the prefix *sr*. Thus, the property of anticipatability used so far is actually *hoist*-anticipatability. We will define a new property *sr*-anticipatability for the purpose of strength reduction. Other data flow properties for strength reduction will be similarly named. Further, for simplicity we omit the prefix *hoist* in contexts where no confusion between the two categories of data flow properties is likely to arise.

sr-properties

An expression e is *sr-anticipatable* at a point in the program if on all paths starting at that point there occurs a computation of e, preceded by at most one update definition of an operand of e, and not preceded by a redefinition of an operand of e. An expression e is *sr-available* at a point in the program if on all paths from the start node to that point there occurs a computation of e, succeeded by at most one update definition of an operand of e, and not succeeded by a redefinition of any operand of e. An expression e is *sr-safe* at a point in the program if it is sr-anticipatable or sr-available at that point.

It should be noted that all update computations succeeding (preceding) an occurrence of an expression e in a program node can be combined into a single update computation. For example, the effect of update computations necessitated by the definitions $x := x+1$ and $x := x + 2$ succeeding (preceding) an occurrence of $x * 5$ can be represented by a single update computation $T_{x*5} := T_{x*5} + 15$. In other words, instead of placing update computations for each definition individually, we can insert a single update computation for a fictitious definition $x := x + 3$. This approach makes it possible to replace *sr*-properties of a *local* significance (namely, *sr*-empty, local *sr*-availability and local *sr*-anticipatability) by their X-counterparts which are both less restrictive and easier to compute. We will use this simplification throughout this paper; hence, the words "update definition" should be understood to mean an update definition in the original program or a fictitious update definition assumed for the purpose of this simplification.

Definition 3.2 *An occurrence of an expression e in a node k is an* sr-eliminatable occurrence *if e is locally X-anticipatable in k and there exists a path $[i \dots k]$ such that :*

(a) e is locally X-available at the exit of i,

(b) All nodes on the path $(i \dots k)$ are X-empty with respect to e, and

(c) e is sr-safe at the exit of each node on the path $[i \dots k)$.

An expression e is *sr-eliminatable from a node k* iff k contains an *sr*-eliminatable occurrence of e or a computation of e placed at the exit of k would be an *sr*-eliminatable occurrence. In figure 2(a), the expression $i * 5$ is sr-eliminatable from node 7 due to the presence of the paths $7-8-10-6-7$, $9-10-6-7$ and $2-3-5-6-7$.

We can now state the desired placement for strength reduction. The placement of recomputations is dictated by criteria similar to DPH–1 and DPH–2, with the terms available, anticipatable and eliminatable replaced by their sr counterparts and the term empty replaced by X-empty. We call these criteria DPS–1 and DPS–2 respectively. In addition we have two criteria, DPS–3 and DPS–4 for the placement of update computations.

Placement of recomputations

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

a. e is not sr-available at the exit of node l,

b. e is not sr-eliminatable from node l, and

c. All paths starting at the exit of l have a prefix $[l \ldots k]$ such that k contains an sr-eliminatable occurrence of e, and for all nodes j on the path $(l \ldots k)$, e is sr-eliminatable from j and j is X-empty with respect to e.

[DPS–2] A recomputation of e is placed in a synthetic node lm on the edge (l, m) iff

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

b. e is not sr-available at the exit of node l, and

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

Placement of update computations

Placement of update computations is performed after the placement of recomputations. This fact simplifies the statement of the placement criteria for update computations by enabling a uniform treatment of paths along which occurrences of e existed in the original program, and paths along which such occurrences are inserted by DPS–1,2. To reflect this fact, we use the words 'in the optimized program' in DPS–3.

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

a. A locally X-available occurrence of e (original computation or inserted recomputation) exists in l[4].

b. k contains an sr-eliminatable occurrence of e.

c. All nodes on the path $(l \ldots k)$ are X-empty with respect to e.

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

For an illustration of these criteria consider again the expression $i * 5$ in figure 2(a). It may be verified that node 4 satisfies DPS–1, node 8 satisfies DPS–3 and node 7 satisifies DPS–4.

Note that the above criteria use the notion of sr-safety for placement of recomputations as follows. The use of sr-anticipatability serves to limit the number of update computations inserted along a path. The motivation for using sr-availability can be understood by observing that it is always profitable to hoist a recomputation from a point where it is sr-available (because on all paths leading to that point a recomputation would be replaced by at most one update computation). Thus sr-safety at all points on a path along which e is hoisted implies that there can be at most two update computations for e along a path $[l \ldots k]$ in the optimized program.

With this placement the set of program paths ending on a deleted computation of e may be classified as follows :

1. the set of paths along which no computations are inserted.
2. the set of paths along which only one recomputation is inserted.
3. the set of paths along which only one update computation is inserted.
4. the set of paths along which two update computations are inserted.
5. the set of paths along which one recomputation and one update computation is inserted.

Paths of type 1, 3 and 4 are assumed to result in the reduction of execution time, and the placement criteria ensure the presence of at least one such path in the optimized program. The presence of a type 5 path will result in some increase in execution time, but it may be offset by the reduction in execution times of the paths of types 1–4. Experimental results in section 6 show that this expectation is satisfied in practice.

[4] l may be an original program node or a synthetic node.

3.3 Properties of the placement

Let the placement of computations be performed as specified in section 3.2. All *sr*-eliminatable occurrences then become redundant and can be eliminated. This is proved in theorem 3.1 below. Theorems 3.2–3.5 prove some important properties of the optimization. Theorem 3.6 proves the correctness of hoisting large expressions along with their constituent subexpressions.

Lemma 3.1 *After the insertion of recomputations, the value of an expression e becomes X-available at the entry of all nodes containing sr-eliminatable occurrences of e.*

Proof :

If possible, let there be a node k containing an *sr*-eliminatable occurrence of e such the value of e is not X-available at the entry of k after the placement of recomputations. This implies the existence of a path $[i \dots k]$ such that

1. i is the initial node or i contains a redefinition of an operand of e,
2. All nodes on the path $(i \dots k)$ are X-empty with respect to e, and
3. No recomputation of e is inserted in any node (original or synthetic) on the path $[i \dots k)$.

Let l be the predecessor of k on the path $[i \dots k]$. Now, e is not X-available (and hence not *sr*-available) at the exit of l. Hence l satisfies DPS–2(b) and DPS–2(c). Since e is not inserted in the synthetic node lk, it must be the case that l violates DPS–2(a), that is l must satisfy DPS–1(c) with respect to e. From the above, l also satisfies DPS–1(a) with respect to e. This, with the fact that e is not inserted in node l, implies that l violates DPS–1(b), that is e must be *sr*-eliminatable from l. Continuing this argument for the predecessor of l on the path $[i \dots k]$, we finally conclude that e is *sr*-eliminatable from node i. Hence, i cannot be the initial node. Also, if i contains a redefinition of an operand of e, then a computation of e placed at the exit of node i would not be locally X-anticipatable, and hence would not be *sr*-eliminatable from i (definition 3.2) — a contradiction. □

Theorem 3.1 (Correctness) *After the placement of recomputations and update computations of an expression e, all sr-eliminatable occurrences of e become redundant.*

Proof :

According to lemma 3.1, all paths in the optimized program ending on an *sr*-eliminatable occurrence of e in a node k would have a suffix $[i \dots k]$ such that

1. i contains a locally X-available occurrence of e (original computation or inserted computation), and

2. All nodes on the path $(i \ldots k)$ are X-empty with respect to e.

According to DPS–3, update computations of e will be placed at the exit of any node l on the path $[i \ldots k)$ having a downward exposed update definition of an operand of e. Let r be the first such node on the path $[i \ldots k)$. Clearly, after the insertion of an update computation of e in node r, e will become available at the exit of r. Using a simple inductive argument we can conclude that e becomes available at the exit of all nodes on the path $[i \ldots k)$. Since this is true for all paths ending on node k, e becomes available at the entry of k. Moreover, if k contains an update definition of an operand of e before the *sr*-eliminatable occurrence of e, then an update computation of e would be placed at the entry of k (DPS–4). The desired result follows. □

Theorem 3.2 (Safety) *On every path starting at an inserted recomputation of an expression e there exists a redundant occurrence of e.*

Proof :

Follows from DPS–1(c) and DPS–2(c). □

Theorem 3.3 (Completeness) *The optimized program does not contain any sr-eliminatable occurrences of a candidate expression.*

Proof :

Let an *sr*-eliminatable occurrence of a candidate expression e exist in the optimized program. This occurrence cannot be an original computation of e, because all *sr*-eliminatable occurrences of e have been removed. Also this occurrence of e cannot be a computation of e inserted in an original program node, because according to DPS–1(b) e is not *sr*-eliminatable from such a node.

So, let this *sr*-eliminatable occurrence of e be a computation inserted in a synthetic node lm satisfying DPS–2. From definition 3.2, there exists a path $[i \ldots lm]$ such that i contains a locally X-available occurrence of e, all nodes on the path $(i \ldots lm)$ are X-empty with respect to e and e is *sr*-safe at the exit of all nodes on the path $[i \ldots lm)$. Further, according to DPS–2(b) e is not *sr*-available at the exit of node l. Hence, e must be *sr*-anticipatable at the exit of node l.

This combined with DPS–2(a) implies that there exists a path $[l \ldots k]$ such that k contains a locally X-anticipatable occurrence of e and e is not *sr*-eliminatable from some node r on the path. But clearly, for all nodes j on the path $[l \ldots k)$, e is *sr*-anticipatable (and hence *sr*-safe) at the exit of j. Therefore, e is *sr*-safe at the exit of all nodes on the path $[i \ldots k)$. From definition 3.2 it follows that e is *sr*-eliminatable from all nodes on the path $[l \ldots k]$ — a contradiction. □

Theorem 3.4 (Profitability) *Optimization does not increase the number of recomputations along any path.*

Proof :

It is sufficient to show that on every path starting at an inserted recomputation of a candidate expression e to a redundant occurrence of e, there does not exist another inserted recomputation of e. The desired result then follows from Theorem 3.2.

Let $[i \ldots r \ldots k]$ be a path in the optimized program such that i and r contain inserted recomputations of e, k contains a redundant occurrence of e and no node on the path $(i \ldots k)$ contains a redundant occurrence of e. From DPS–1(c) and DPS–2(c) it follows that the occurrence of e in r is *sr*-eliminatable, which contradicts Theorem 3.3. □

Definition 3.3 *A* redundant hoist *is a hoist of an expression e into a node l when its insertion into some node(s) $i \in descendant(l)$ would have satisfied the safety, completeness and profitability properties (theorems 3.2–3.4).*

Theorem 3.5 *No redundant hoists occur.*

Proof :

Let an expression e be redundantly hoisted from node k to node l. From DPS–1(c) and DPS–2(c) it follows that e is *sr*-eliminatable from all nodes on the path $(l \ldots k]$. Hence an insertion of e in any of these nodes would have constituted an *sr*-eliminatable partial redundancy, thus violating the condition of completeness — a contradiction. □

Theorem 3.6 *Let e be a large expression and e_1 be a subexpression of e. Then e_1 is available at all points where a recomputation of e is inserted.*

Proof :

Let e be inserted in a node l of the optimized program. Then from theorem 3.2, all paths starting at node l have a prefix $[l \ldots k]$ such that k contains an *sr*-eliminatable occurrence of e (and hence of e_1) and all nodes j on the path $(l \ldots k)$ are X-empty with respect to e (and hence with respect to e_1).

From definition 3.2, it follows that e_1 is *sr*-eliminatable from a node j if e is *sr*-eliminatable from j. Thus, no recomputation of e_1 would get inserted along the path

$(l \dots k)$, because such an insertion would constitute an *sr*-eliminatable occurrence, and violate theorem 3.3. Two cases arise :

1. e_1 is not *sr*-eliminatable from node l. Then, DPS–1(b) implies that e_1 is inserted in node l. In this case, the policy of inserting computations of large expressions *after* the computations of their constituent subexpressions ensures the required result.

2. e_1 is *sr*-eliminatable from node l. Then, theorem 3.3 implies that no recomputation of e_1 is inserted in node l. In this case, lemma 3.1 implies that all paths ending on node l have a suffix $[i \dots l]$ such that node i contains a recomputation (inserted or original) of e_1 and all nodes on the path $[i \dots l]$ are X-empty with respect to e_1. Moreover, theorem 3.1 implies that e_1 is available at the entry of node l. If l is empty with respect to e_1, then the required result is obviously true. On the other hand, if l contains an update definition of an operand of e_1, then according to DPS–3, an update computation of e_1 would be inserted in l. Again, the policy of inserting computations of e after those for e_1, ensures the required result.

□

4 The SRLE algorithm

We use the partial redundancy elimination framework of Morel and Renvoise [4] to implement the desired placement of section 3.2. Since the placement involves insertion of computations in synthetic blocks placed along graph edges, we model our equations after the edge placement algorithm of Dhamdhere [5]. The local and global data flow properties are described in tables 1 and 2. Table 3 shows the data flow problems used in the algorithm. The SRAVIN/OUT problem computes sr-availability of an expression, while the SRPPIN/OUT problem computes the possibility of placing a recomputation of an expression subjected to optimization at the entry/exit of a node. The properties INSERT and UINSERT indicate insertion of recomputations and update computations of expression e in a node, while SAVE marks nodes where the value of an existing evaluation of e should be saved for use in the elimination of a partial redundancy. REDUND marks nodes containing evaluations of e which are redundant following the optimization.

Section 4.1 describes the solution approach for obtaining the desired fixed point of the data flow equations.

Algorithm 4.1 :

1. Solve the data flow equations in table 3.

2. For each expressions e, create an expression variable T_e.

3. For each expression e do :

   (a) for each i with $\text{INSERT}_i(e)$: Insert the recomputation $T_e := e$ at the exit of program node i.
   for each ji with $\text{INSERT\_SYNTH}_{ji}(e)$: Insert the statement $T_e := e$ in synthetic node ji.

   (b) for each i with $\text{UINSERT\_EXIT}_i(e)$: Insert an update computation $T_e := T_e + g$, where g is an appropriate constant, at the exit of node i.
   for each i with $\text{UINSERT\_ENTRY}_i(e)$: Insert an update computation $T_e := T_e + g$, where g is an appropriate constant, at the entry of node i.

   (c) for each i with $\text{REDUND}_i(e)$: Replace the first computation of e in node i by T_e.

   (d) for each i with $\text{SAVE}_i(e)$: Insert the statement $T_e := e$ before the last computation of e in node i, and replace that computation by T_e.

In step 3 of the algorithm, it is important that recomputations for a large expression e, namely $T_e := T_{e_1}\, op\, T_{e_2}$, be inserted *after* any inserted recomputations or update computations of subexpressions of e, namely $T_{e_1} = e_1$ or $T_{e_1} = T_{e_1} + g$. Detailed proofs of conformance to the placement criteria of section 3.2 are given in the appendix.

Note that for expressions which are not candidates for strength reduction, setting UPDATE_i = False, $\text{XTRANSP}_i = \text{TRANSP}_i$, $\text{XANTLOC}_i = \text{ANTLOC}_i$ and $\text{XCOMP}_i = \text{COMP}_i$ for all nodes i achieves simple hoisting.

A common concern regarding the composite optimization algorithms is their space requirement. A straightforward implementation of the SRLE algorithm would require bitvector storage for 7 local properties and 20 global properties (assuming the IN and OUT of each global property in table 2 to be stored seperately). However, it is possible to optimize storage by exploiting the limited lifetimes of most of the properties. For example, the storage space for the AVIN/OUT properties can be reused after computing MXAVIN/OUT. This optimization reduces the storage requirements of the global properties to only 8 bitvectors.

Table 1: Local properties

(*Note* : All properties are described with respect to a single expression e.)

ANTLOC_i	:	node i contains an occurrence of e, not preceded by a definition of any of its operands.
COMP_i	:	node i contains an occurrence of e, not followed by a definition of any of its operands.
TRANSP_i	:	node i does not contain definition(s) of any operand(s) of e.
UPDATE_i	:	node i contains an update definition of an operand of e not followed by an occurrence of e or a redefinition of any of its operands.
$\mathrm{XANTLOC}_i$	:	node i contains an occurrence of e, not preceded by a redefinition of any of its operands.
XCOMP_i	:	node i contains an occurrence of e, not followed by a redefinition of any of its operands.
$\mathrm{XTRANSP}_i$	:	node i does not contain redefinition(s) of any operand(s) of e.

Table 2: Global Properties

(*Note* : All properties are described with respect to a single expression e.)

$\mathrm{AVIN}_i/\mathrm{AVOUT}_i$	:	e is available at the entry/exit of node i.
$\mathrm{ANTIN}_i/\mathrm{ANTOUT}_i$	:	e is anticipatable at the entry/exit of node i.
$\mathrm{SRAVIN}_i/\mathrm{SRAVOUT}_i$	:	e is *sr*-available at the entry/exit of node i.
$\mathrm{SRANTIN}_i/\mathrm{SRANTOUT}_i$	:	e is *sr*-anticipatable at the entry/exit of node i.
$\mathrm{SRPPIN}_i/\mathrm{SRPPOUT}_i$	:	a recomputation may be placed at the entry/exit of node i.
$\mathrm{HOISTIN}_i/\mathrm{HOISTOUT}_i$	:	e is hoisted past the entry/exit of node i.
$\mathrm{UINSERT\_EXIT}_i$	:	an update computation should be inserted at the exit of node i.
$\mathrm{UINSERT\_ENTRY}_i$	:	an update computation should be inserted at the entry of node i.
INSERT_i	:	a recomputation should be inserted at the exit of node i.
$\mathrm{INSERT\_SYNTH}_{ji}$	:	a recomputation should be inserted in a synthetic node along edge (j, i).
REDUND_i	:	the first occurrence of e in node i is redundant and may be deleted.
SAVE_i	:	the value of the last computation of e in node i should be saved in expression variable T_e.

Table 3: Data Flow Equations

$$
\begin{aligned}
\mathrm{AVOUT}_i &= \mathrm{COMP}_i + \mathrm{TRANSP}_i \cdot \mathrm{AVIN}_i && (1)\\
\mathrm{AVIN}_i &= \prod_{p \in pred(i)} \mathrm{AVOUT}_p && (2)\\
\mathrm{ANTIN}_i &= \mathrm{ANTLOC}_i + \mathrm{TRANSP}_i \cdot \mathrm{ANTOUT}_i && (3)\\
\mathrm{ANTOUT}_i &= \prod_{s \in succ(i)} \mathrm{ANTIN}_s && (4)\\
\mathrm{SRAVOUT}_i &= \mathrm{XCOMP}_i + \mathrm{XTRANSP}_i \cdot \mathrm{AVIN}_i + \\
&\quad \mathrm{TRANSP}_i \cdot \mathrm{SRAVIN}_i && (5)\\
\mathrm{SRAVIN}_i &= \prod_{p \in pred(i)} \mathrm{SRAVOUT}_p && (6)\\
\mathrm{SRPPIN}_i &= [\mathrm{XANTLOC}_i + \mathrm{SRPPOUT}_i \cdot (\mathrm{TRANSP}_i + \\
&\quad \mathrm{XTRANSP}_i \cdot \mathrm{ANTOUT}_i)] \cdot \\
&\quad \sum_{p \in pred(i)} (\mathrm{SRAVOUT}_p + \mathrm{SRPPIN}_p \cdot \neg\mathrm{XANTLOC}_p) && (7)\\
\mathrm{SRPPOUT}_i &= \prod_{s \in succ(i)} \mathrm{SRPPIN}_s && (8)\\
\mathrm{REDUND}_i &= \mathrm{SRPPIN}_i \cdot \mathrm{XANTLOC}_i && (9)\\
\mathrm{INSERT}_i &= \mathrm{SRPPOUT}_i \cdot (\neg\mathrm{SRPPIN}_i + \neg\mathrm{XTRANSP}_i) \cdot \\
&\quad \neg\mathrm{SRAVOUT}_i && (10)\\
\mathrm{INSERT\_SYNTH}_{ji} &= \mathrm{SRPPIN}_i \cdot \neg\mathrm{SRPPOUT}_j \cdot \neg\mathrm{SRAVOUT}_j && (11)\\
\mathrm{HOISTIN}_i &= \mathrm{REDUND}_i + \mathrm{HOISTOUT}_i \cdot \neg\mathrm{INSERT}_i \cdot \neg\mathrm{XCOMP}_i && (12)\\
\mathrm{HOISTOUT}_i &= \sum_{s \in succ(i)} \mathrm{HOISTIN}_s \cdot \neg\mathrm{INSERT\_SYNTH}_{is} && (13)\\
\mathrm{UINSERT\_EXIT}_i &= \mathrm{UPDATE}_i \cdot \mathrm{HOISTOUT}_i \cdot \neg\mathrm{INSERT}_i && (14)\\
\mathrm{UINSERT\_ENTRY}_i &= \mathrm{REDUND}_i \cdot \neg\mathrm{ANTLOC}_i && (15)\\
\mathrm{SAVE}_i &= \mathrm{HOISTOUT}_i \cdot \mathrm{XCOMP}_i \cdot \\
&\quad (\neg\mathrm{HOISTIN}_i + \neg\mathrm{XTRANSP}_i) && (16)
\end{aligned}
$$

4.1 Solution of the data flow equations

The data flow problems AVIN/OUT, ANTIN/OUT, SRAVIN/OUT and HOISTIN/OUT in table 3 are standard unidirectional problems and can be solved using any of the standard methods. The data flow problem of SRPPIN/OUT (equations (7)-(8)) is however a bidirectional problem. A similar bidirectional problem arising in the edge placement algorithm (EPA) has been solved in [5, 12] by decomposing it into two unidirectional data flow problems[5]. Using a similar approach, we decompose the SRPPIN/OUT problem into the following two unidirectional problems.

(a) **The Approximation Problem**

$$\begin{aligned}\mathrm{SRPPIN}_i^{apx} &= \mathrm{XANTLOC}_i + \mathrm{SRPPOUT}_i^{apx} \cdot (\mathrm{TRANSP}_i + \\ &\quad \mathrm{XTRANSP}_i \cdot \mathrm{ANTOUT}_i) \qquad (17)\\ \mathrm{SRPPOUT}_i^{apx} &= \prod_{s \in succ(i)} \mathrm{SRPPIN}_s^{apx} \qquad (18)\end{aligned}$$

(b) **The Refinement Problem**

$$\mathrm{SRPPIN}_i = \mathrm{SRPPIN}_i^{apx} \cdot \sum_{p \in pred(i)} (\mathrm{SRAVOUT}_p + \mathrm{SRPPIN}_p \cdot \neg \mathrm{XANTLOC}_p) \qquad (19)$$

Since the basic data flow involved in hoisting is *backwards*, the approximation problem incorporates only the backward data flow of equations (7)-(8); we eliminate the forward dependence of SRPPIN of a node on its predecessors by dropping the $\sum$ term in the SRPPIN equation. The resulting data flow problem resembles the classical data flow problem of very busy expressions.

The refinement problem merely incorporates the effect of the $\sum$ term of the SRPPIN_i equation (equation (7)). It is thus a *forward* data flow problem. One aspect of the original data flow, namely the influence of the final value of SRPPIN_s on the value of $\mathrm{SRPPOUT}_i$ (equation (8)), has to be incorporated after refinement. This is done by computing the final values of SRPPOUT for all nodes of the graph in a single pass over the graph.

The decomposition of the bidirectional data flow into two unidirectional data flows is made possible by the following peculiarity of the data flow : SRPPIN_i = false might make SRPPIN_s = false for some successor s of node i. This, in turn, makes $\mathrm{SRPPOUT}_i$

[5]Dhamdhere, Khedker [10, 11] subsequently stated the conditions on the feasibility of decomposing a bidirectional data flow problem into a sequence of unidirectional data flow problems

= false. However, SRPPOUT_i = false cannot influence SRPPIN_i, because it is already false. (Dhamdhere and Khedker [10, 11] present this fact formally using the notion of *information flow paths.*) Hence the appromixation and refinement data flow problems need to be solved only once.

It is interesting to compare equation (19) with the refinement approach of Dhamdhere and Patil [12], which involves solving an equation of the form

$$\text{SRPPIN}_i^{ref} = \sum_{p \in pred(i)} (\text{SRAVOUT}_p + \text{SRPPIN}_p^{ref} \cdot \neg\text{XANTLOC}_p) \tag{20}$$

with SRPPIN_i^{ref} initialized to SRPPIN_i^{apx}. This approach gives a fixed point in which a form of redundant hoisting, known as "hoisting through the loop effect" persists. (This effect is illustrated in section 4.2.) The refinement equation is also not a standard unidirectional data flow problem due to the initialization of SRPPIN_i^{ref} to SRPPIN_i^{apx} (instead of initializing to $false$ as required by the confluence operator $\sum$). Dhamdhere and Patil [12] give an algorithm based on a variant of interval analysis to eliminate such hoisting.

In our method, we use the result of the approximation step, that is SRPPIN_i^{apx}, as an additional term in the data flow equation of the refinement step, rather than as the initial value of SRPPIN_i (see equation (19)). This not only avoids redundant hoisting as explained in section 4.2, it also yields a standard unidirectional data flow problem. Thus, both approximation and refinement problems can be solved by any of the well known algorithms for data flow analysis.

4.2 An Example

Table 4 shows the data flow properties for the three candidate expressions $i * 5$, $j * 10$ and $i * 5 + j * 10$ in the example of fig. 2. The salient points of the optimization are as follows :

1. Simple hoisting occurs for the expression $j * 10$, and strength reduction for the large expression $i * 5 + j * 10$ as well as its subexpression $i * 5$ (see fig. 2(b)). The update computations for $i * 5$ inserted in nodes 7 and 8 are assumed to be removed during dead code elimination.

2. The solution approach of section 4.1 avoids redundant hoisting. Hence the expressions $j * 10$ and $i * 5 + j * 10$ are hoisted only as far as node 5. Most earlier algorithms for hoisting or composite hoisting and strength reduction [4, 5, 2, 3]

Table 4: Data Flow Properties for the example in figure 2

(*Note* : Each entry is a set of nodes for which a property is true.)

Property	Values of the properties for		
	$i*5$	$j*10$	$i*5+j*10$
SRPPIN^{apx}	$\{2,3,5,6,7,9\}$	$\{2,3,5,6,7,9\}$	$\{2,3,5,6,7,9\}$
SRPPOUT^{apx}	$\{2,3,4,5,6\}$	$\{2,3,4,5,6\}$	$\{2,3,4,5,6\}$
SRPPIN	$\{3,5,6,7,9\}$	$\{6,7,9\}$	$\{6,7,9\}$
SRPPOUT	$\{2,3,4,5,6\}$	$\{5,6\}$	$\{5,6\}$
REDUND	$\{7,9\}$	$\{7,9\}$	$\{7,9\}$
INSERT	$\{4\}$	$\{5\}$	$\{5\}$
INSSYNTH	$\{\}$	$\{\}$	$\{\}$
HOISTIN	$\{3,5,6,7,8,9,10\}$	$\{6,7,8,9,10\}$	$\{6,7,8,9,10\}$
HOISTOUT	$\{2,3,4,5,6,7,8,9,10\}$	$\{5,6,7,8,9,10\}$	$\{5,6,7,8,9,10\}$
UINSERT-EXIT	$\{8\}$	$\{\}$	$\{8\}$
UINSERT-ENTRY	$\{7\}$	$\{\}$	$\{7\}$

would redundantly hoist these computations through the loop $\{3\}$ to place them in node 2 (and node 4). This problem, called the "hoisting thorough the loop effect", occurs as follows [12] : Consider the expression $j*10$. If eq. (20) is used instead of eq. (19), SRPPIN$_3$ and SRPPOUT$_3$ would be initialized to SRPPIN$_3^{apx}$ and SRPPOUT$_3^{apx}$, both of which are *true*. SRPPIN$_3$ now sustains itself due to the $\sum$ confluence operator. Consequently SRPPIN$_3$ and SRPPIN$_5$ remain *true*. In our method of refinement, SRPPIN^{apx} is simply a constant of the refinement problem (eq. 19) and SRPPIN$_i$/SRPPOUT$_i$ would be initialized to $false$ due to the $\sum$ confluence operator. Hence SRPPIN$_3$ remains $false$.

3. The type of paths ending on redundant computations of $i*5+j*10$ are :

    - Paths along which only one update computation is inserted ($7-8-10-6-9$ and $9-10-6-7$).
    - Paths along which two update computations are inserted ($7-8-10-6-7$).
    - Paths along which one recomputation and one update computation is inserted ($5-6-7$).
    - Paths along which only one recomputation gets inserted ($5-6-9$).
    - Paths along which no computations get inserted ($9-10-6-9$).

5 Comparison with related work

Related work can be classified into two groups of papers. The first group concerns the properties of code placement performed during partial redundancy elimination. The second group concerns algorithms for strength reduction — both classical loop based algorithms and composite algorithms based on the framework of partial redundancy elimination.

Classical redundancy elimination algorithms focus on common subexpression elimination for lexically identical expressions. This approach cannot optimize algebraically equivalent expressions, for example $b*c$ and $a*c$ in $a := b;\ \ldots\ b*c\ \ldots\ a*c$. Cocke and Schwartz [19] proposed value numbering to identify algebraically equivalent expressions. Rosen *et al* [20] report a redundancy elimination algorithm using value numbering.

Morel and Renvoise [4] were the first to propose a composite hoisting and common subexpression elimination algorithm, using a framework for the elimination of partial redundancies in a program. Their approach only considers lexically identical occurrences of expressions for the purpose of optimization, thus missing out on the optimization of algebraically equivalent expressions. All algorithms based on their partial redundancy elimination framework inherit this feature.

Dhamdhere [5] proposed the *edge placement* approach to overcome the major drawbacks of the Morel-Renvoise algorithm, namely redundant hoisting and incomplete partial redundancy elimination. Dhamdhere and Patil [12] proposed an elimination approach to the solution of data flows involved in edge placement. However as mentioned in section 4.1, their method requires a fairly complex variant of interval analysis to eliminate all redundant hoisting. This is achieved far more simply by SRLE.

Knoop et al. [21] perform extensive edge splitting and use the node placement model as in [4]. This approach enables them to decompose the placement algorithm into a series of unidirectional problems. The solution they obtain is essentially the same as the one obtained by SRLE as far as expressions subject to simple hoisting are concerned, except for some differences due to edge splitting. Consider fig. 3(a) wherein the partially redundant computations of expression e in nodes 4 and 5 are to be hoisted. Edge splitting followed by node placement results in computations of e getting inserted into nodes 6 and 7 introduced by edge splitting (see fig. 3(b)). On the other hand, our approach based on edge placement creates synthetic nodes only where necessary and results in insertion of a single computation of e in node 3. From the practical viewpoint their algorithm suffers from two other drawbacks — the overhead of edge splitting and the inapplicability of the algorithm (as stated) to flow graphs in which nodes represent multi-statement basic

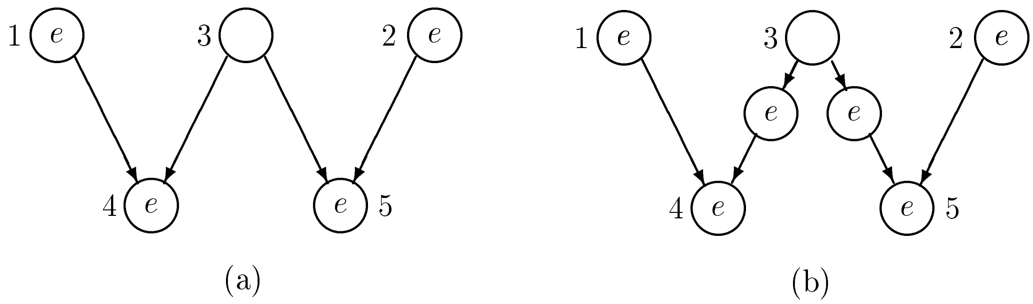


Figure 3: Difference between edge splitting and edge placement

blocks.

Drechsler and Stadel [22] refine the work of Knoop et al. [21] to make it more practical. Their approach does not require edge splitting to precede data flow analysis, permits multi-statement basic blocks and involves simpler and fewer data flow problems. The storage requirements and computations involved in their approach are as follows : 3 local properties are required per basic block, 5 global properties are computed through 3 unidirectional data flows and 4 other properties are computed for the purpose of code placement. They also require an extra pass to avoid the problem shown in fig. 3. On the other hand, our method applied to simple hoisting (where the X-properties and *sr*-properties reduce to the simpler *hoist*-properties) requires 3 local properties per basic block, 4 global properties are computed through 3 unidirectional data flows and 3 other properties are computed for the purpose of code placement. Further, the problem of fig. 3 does not arise because node-and-edge placement is directly performed.

The classical strength reduction algorithms work on subgraphs of the program flow graph satisfying certain properties. For example, the strength reduction algorithms in [8, 9] identify *induction variables* and *region constants* of single-entry strongly-connected subgraphs, and perform strength reduction of multiplicative expressions involving these. In contrast, partial redundancy elimination techniques on which the composite algorithms are based make no assumptions about the structure of the program flow graph; they deal exclusively with data flow properties of flow graph paths. Hence the classical approach has two limitations in comparison to the composite algorithms. First, they require an additional step to detect these control structures. Second, some opportunities for strength reduction exploited by the composite algorithms are lost by confining the scope of optimization to subgraphs with specific properties [2].

SRLE and NCHSA [2] provide guarantees concerning the maximum number of update computations inserted along any path. It is difficult to see how such guarantees can be

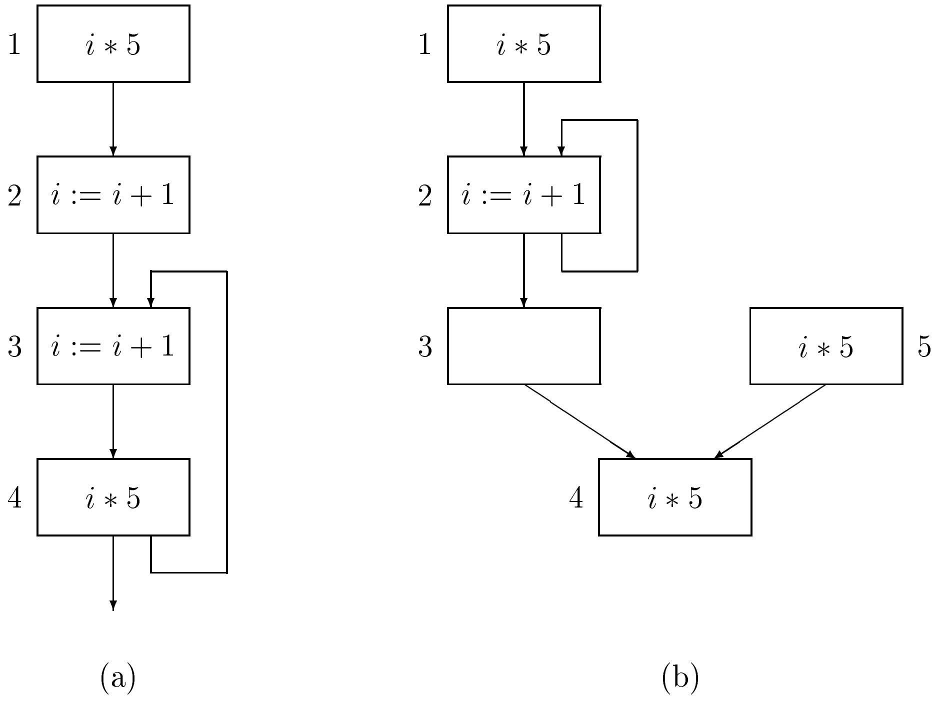


Figure 4: Examples of improvements over NCHSA

incorporated into the classical algorithms without resorting to ad-hoc means, for example limiting the total number of update definitions in a loop. This approach would be unduly restrictive. On the other hand, the classical algorithms can consider a greater variety of expressions as candidates for strength reduction than the composite algorithms. In particular, they are able to strength reduce multiplicative expressions containing arbitrary region constants rather than just simple constants [8, 9]. Composite algorithms do not consider individual regions and cannot strength reduce such expressions. A more detailed discussion of this issue may be found in [2].

Dhamdhere [1] and Dhamdhere and Isaac [7] are the early works on composite hoisting and strength reduction. Joshi and Dhamdhere [2] were the first to incorporate strength reduction in the Morel-Renvoise framework. Their policy of strength reduction however gave rise to certain problems which were mentioned in section 3.2. Moreover the algorithm still suffered from the drawbacks of the Morel-Renvoise algorithm mentioned earlier.

Dhamdhere [3] reports an algorithm (NCHSA) to overcome the problems in [2] using a policy similar to the one described in this paper. However, the NCHSA algorithm does not always implement the policy in a desirable manner. For example it fails to perform any optimization at all in the program graph of figure 4(a). SRLE optimizes the program by inserting update computations in nodes 2 and 3 and deleting the computations in node 4. Thus not only is the computation of $i * 5$ hoisted out of the loop, but profitability from further hoisting is also exploited. As another example consider figure 4(b). Again NCHSA will not optimize this, but SRLE will do so by hoisting the partially redundant computation of $i * 5$ to node 3. Another shortcoming of NCHSA is that its equations cannot be used to optimize large expressions. More precisely, lemma 3.6 in section 3.3 does not hold for the equations. Moreover, NCHSA still suffers from the drawback of the "hoisting through the loop" effect and, being based on the node placement model, cannot remove all eliminatable partial redundancies. SRLE uses a placement model which ensures completeness of optimization without any of the penalties associated with edge splitting. Use of the notions of *sr*-availability and *sr*-anticipatability eliminates the drawbacks of earlier works [2, 3] of composite hoisting and strength reduction.

Classical strength reduction algorithms can be extended to incorporate the strength reduction of large expressions in exactly the same manner as in SRLE, namely by considering expressions of the kind $e_1 \pm e_2$ where e_1 and e_2 are themselves candidate expressions. However, these algorithms cannot easily incorporate simple hoisting. In that sense SRLE is the first practical composite hoisting and strength reduction algorithm. Section 6 describes the practical advantages of the strength reduction of large expressions.

6 Experimental results

Table 5 summarizes the comparative experimental performance of the SRLE algorithm and the NCHSA algorithm when applied to a set of Fortran subroutines in the LINPACK package. Note that the NCHSA algorithm was implemented on elementary expressions as candidates, and the unidirectional and bidirectional data flows described in [3] were implemented using a round robin iterative algorithm. In contrast, SRLE was implemented to handle *large* expressions, and the bidirectional data flow of the $\mathrm{SRPPIN}_i/\mathrm{SRPPOUT}_i$ was solved as a sequence of unidirectional data flows as described in section 4.1. Thus, the significant points of comparison are the comparative effectiveness of optimization performed by the two algorithms, and their solution costs.

For comparing the effectiveness of the optimizations performed by the two algorithms, we use the approach of comparing the 'static frequency counts' used in much of register assignment work. In this approach the execution cost of a program is the sum of the weighted execution costs of each of its expressions. The weighting factor for an expression is a prediction of its execution frequency during a typical run of the program. We use the weighing factor of w^i, where i is the static loop nesting level of the expression and w is an appropriately chosen constant indicating the likely number of iterations of each loop. We choose $w = 5$ as a very conservative estimate.

Table 5: Experimental results

(*Note* : The weighting factor is assumed to be 5^i — see section 6 for details.)

Program	N	E	SRLE			NCHSA			Improvement factors	
			D_1	I_1	B_1	D_2	I_2	B_2	I^*	B^*
MATGEN	23	45	24	41.6	5469.0	10	29.3	14526.5	1.42	2.65
DGEFA	26	106	16	54.7	8510.0	5	47.9	30336.4	1.14	3.56
DGESL	32	76	32	22.1	10036.0	0	17.6	24074.4	1.26	2.40
DAXPY	19	61	20	23.8	4872.0	6	9.4	7781.5	2.43	1.60
DDOT	18	68	23	24.7	4625.0	11	13.1	9486.5	1.89	2.05
DSCAL	14	33	12	20.8	2258.0	4	7.4	4928.0	2.81	2.18
IDAMAX	29	44	18	50.8	6967.0	10	36.9	13881.5	1.38	1.99
DMXPY	37	263	152	40.0	27209.5	12	31.1	68310.0	1.28	2.51
								Average	1.70	2.37

Notation :

N : the number of basic blocks.

E : the number of elementary expressions in the program.

D : the number of quadruples removed by dead code elimination.

I : the percentage improvement in the estimated execution cost of the program.

B : the average number of bitvector operations performed per application of the algorithm.

I^* : the improvement factor in the effectiveness of optimization $= I_1/I_2$.

B^* : the improvement factor in the number of bitvector operations $= B_2/B_1$.

The following expectations were borne out by the results (See Table 5) :

1. The SRLE algorithm yields a significant improvement in the effectiveness of optimization[6].

2. Opportunities for dead code elimination increase when strength reduction is applied to large expressions.

3. The simpler data flow equations of SRLE and the solution of the bidirectional problem using decomposition result in a substantial decrease in the number of bitvector operations performed and hence in the execution cost of the optimizer.

7 Summary

The SRLE algorithm performs strength reduction of a large expression as a single entity using the framework for partial redundancy elimination [4, 5, 6]. It uses the ideas of composite hoisting and strength reduction [1, 2, 3], thereby subsuming the conventional optimizations of code hoisting, common subexpression elimination, loop invariant movement and strength reduction. This approach to the optimization of large expressions leads to more dead code elimination than in conventional strength reduction algorithms.

Apart from extending composite hoisting and strength reduction to large expressions, the formulation of the strength reduction problem used in SRLE eliminates several suboptimalities of previously published work in partial redundancy based strength reduction [2, 3]. The notion of sr-safety used in the algorithm provides an elegant solution to the problem of restricting the number of update computations inserted along any path in the optimized program. It also eliminates the failure of earlier algorithms [2, 3] to optimize certain profitable situations.

A solution technique for the bidirectional data flows involved in the SRLE algorithm is also proposed. It provides a simple and elegant solution to the problem of 'redundant hoisting' [5] suffered by many algorithms using the partial redundancy elimination framework. In section 5, this solution technique is shown to be superior to the ones proposed in [21, 22].

Acknowledgements

The authors wish to thank the referees for many valuable suggestions.

[6]Since the algorithm performs loop optimization, its effectiveness increases with increasing values of w.

References

[1] Dhamdhere, D. M. : On algorithms for operator strength reduction, *Communications of the ACM*, **22**(5):311-312, 1979.

[2] Joshi, S. M. and D. M. Dhamdhere : A composite algorithm for strength reduction and code movement, *International Journal of Computer Mathematics*, part I - **11**(1):21-44, part II - **11**(2):111-126, 1982.

[3] Dhamdhere, D. M. : A new algorithm for composite hoisting and strength reduction, *International Journal of Computer Mathematics*, **27**(1):1-14, 1989.

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

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

[6] Dhamdhere, D. M. : Practical adaptation of global optimization algorithm by Morel & Renvoise, *ACM Transactions on Programming Languages & Systems*, **13**(2):291-294, 1991.

[7] Dhamdhere, D. M. and J. R.Isaac : A composite algorithm for strength reduction and code movement, *International Journal of Computer and Information Sciences*, **9**(3):243-273, 1980.

[8] Cocke, J. and K. Kennedy : An algorithm for the reduction of operator strength, *Communications of the ACM*, **20**(11):850-856, 1977.

[9] Allen, F. E., J. Cocke and K. Kennedy : Reduction in operator strength, in *Program Flow Analysis : Theory and Applications* (pages 79-101), S.Muchnick and N.Jones (eds), Prentice Hall, 1981.

[10] Dhamdhere, D. M. and Uday P. Khedker : Complexity of bidirectional data flow analysis, *Proceedings of the Twentieth ACM Symposium on Principles of Programming Languages* (pages 397-408), 1993.

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

[12] Dhamdhere, D. M. and Harish 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.

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

[14] Kennedy, K. : Safety of Code Movement, *International Journal of Computer Mathematics*, **3**:112-130, 1972.

[15] Schafer, M. : *A Mathematical Theory of Program Optimization*, Prentice Hall, 1973.

[16] Dhamdhere, D. M., B. K. Rosen and F. K. Zadeck : How to analyze large programs efficiently and informatively, *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation* (pages 212-223), 1992.

[17] Joshi, S. M. : Studies in code optimization, M. Tech. dissertation, IIT Bombay, 1980.

[18] Drechsler, K. 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.

[19] Cocke, J. and J. T. Schwartz : *Programming Languages and Their Compilers; Preliminary Notes*, Courant Institute of Mathematical Sciences, New York University, April 1970.

[20] Rosen, B. K., M. N. Wegman and F. K. Zadeck : Global value numbers and redundant computations, *Conference Record of the 15th ACM Symposium on Principles of Programming Languages*, (pages 12-27), Jan. 1988.

[21] Knoop, J., O. Rüthing, B. Steffan, : Lazy Code Motion, *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation* (pages 224-234), 1992.

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

A Properties of the SRLE algorithm

In this section we shall prove the correctness of the SRLE algorithm. This involves showing that the placement realized by the data flow equations in section 4 is identical to that defined by the criteria given in section 3. Lemma A.1 and theorems A.1, A.2 and A.3 achieve this. Proofs of the correctness of equations for availability, anticipatability and *sr*-availability are trivial and hence omitted.

Lemma A.1

(a) *SRPPIN*$_i^{apx}(e) \Longleftrightarrow$*e is sr-anticipatable at the entry of node i.*

(b) *SRPPIN*$_i(e) \Longleftrightarrow$*e is sr-anticipatable at the entry of node i and e is sr-eliminatable from i.*

(c) *SRPPOUT*$_i(e) \Longleftrightarrow$*e is sr-anticipatable at the exit of node i and all paths starting from the exit of i have a prefix* $[i \ldots k]$ *such that k contains an sr-eliminatable occurrence of e, all nodes on the path* $(i \ldots k)$ *are X-empty and e is sr-eliminatable from all nodes on the path* $(i \ldots k]$.

Proof :

(a) From equations (17) and (18) we have $\text{SRPPIN}_i^{apx}(e) \Longleftrightarrow$ all paths starting from the entry of i have a prefix $[i \ldots k]$ such that

1. $\text{XANTLOC}_k(e)$, and
2. $\text{TRANSP}_j(e) + \text{XTRANSP}_j(e) \cdot \text{ANTOUT}_j(e)$ for all nodes j on the path $[i \ldots k)$.

To prove that $\text{SRPPIN}_i^{apx}(e) \Rightarrow \text{SRANTIN}_i(e)$, we shall prove that any path $[i \ldots k]$ of the type described above can have at most one update definition of an operand of e. If possible let there be two nodes r and s on the path containing update definitions and let r precede s on the path $[i \ldots k]$. Then we have $\neg\text{TRANSP}_r(e) \cdot \neg\text{ANTOUT}_r(e)$. Hence r violates condition 2 given above — a contradiction.

Now consider a node i with $\text{SRANTIN}_i(e)$. Then all paths starting from the entry of node i have a prefix $[i \ldots k]$ such that

1. $\text{XANTLOC}_k(e)$,

2. $\mathrm{XTRANSP}_j(e) \cdot \neg\mathrm{XANTLOC}_j(e)$ for all nodes j on the path $[i \ldots k)$, and
3. There is at most one node on the path $[i \ldots k]$ containing an update definition of an operand of e.

To prove that $\mathrm{SRANTIN}_i(e) \Rightarrow \mathrm{SRPPIN}_i^{apx}(e)$ it is sufficient to prove that $\mathrm{TRANSP}_j(e) + \mathrm{XTRANSP}_j(e) \cdot \mathrm{ANTOUT}_j(e)$ for all nodes j on the path $[i \ldots k)$ described above. Let r be the only node on the path $[i \ldots k]$ containing an update definition of e. If possible let $\neg\mathrm{ANTOUT}_r(e)$. Then we must have a path $[r \ldots s]$ such that s is the exit node or $\neg\mathrm{TRANSP}_s(e)$, and $\mathrm{TRANSP}_l(e) \cdot \neg\mathrm{ANTLOC}_l(e)$ for all nodes l on the path $(r \ldots s)$. But then the existence of the path $[i \ldots r \ldots s]$ contradicts $\mathrm{SRANTIN}_i(e)$. Thus we conclude $\mathrm{XTRANSP}_r(e) \cdot \mathrm{ANTOUT}_r(e)$. For all other nodes j on the path $[i \ldots k)$ we have $\mathrm{TRANSP}_j(e)$. The desired result then follows.

(b) $\mathrm{SRPPIN}_i \Longleftrightarrow \mathrm{SRPPIN}_i^{apx}$ and there exists a path $[l \ldots i]$ such that $\mathrm{SRAVOUT}_l$ and for all nodes j on the path $(l \ldots i)$ $\mathrm{SRPPIN}_j^{apx} \cdot \neg\mathrm{XANTLOC}_j$ (eq. 19). From equation (17) $\mathrm{SRPPIN}_j^{apx} \cdot \neg\mathrm{XANTLOC}_j \Rightarrow \mathrm{XTRANSP}_j$. Hence $\mathrm{SRPPIN}_j^{apx} \cdot \mathrm{XTRANSP}_j \cdot \neg\mathrm{XANTLOC}_j$ for all nodes j.

Thus from part (a) we have : $\mathrm{SRPPIN}_i(e) \Longleftrightarrow \mathrm{SRANTIN}_i(e)$ and there exists a path $[l \ldots i]$ such that (i) $\mathrm{SRAVOUT}_l(e)$, (ii) $\mathrm{SRANTIN}_j(e)$ for all nodes j on the path $(l \ldots i)$, and (iii) All nodes j on the path $(l \ldots i)$ are X-empty with respect to e.

From (i)–(iii) and definition 3.2 it follows that $\mathrm{SRPPIN}_i(e) \Longleftrightarrow e$ is *sr*-anticipable at the entry of node i and e is *sr*-eliminatable from node i.

(c) From equation (8), $\mathrm{SRPPOUT}_i(e) \Longleftrightarrow \mathrm{SRPPIN}_s(e)$ for each successor s of i.

$\Longleftrightarrow e$ is *sr*-anticipatable at the exit of i and *sr*-eliminatable from s (from part (b) of the lemma).

$\Longleftrightarrow$each path starting from the exit of node i has a prefix $[s \ldots k]$ such that k contains a locally X-anticipatable occurrence of e, e is *sr*-eliminatable from s and all nodes j on the path $[s \ldots k)$ are such that (i) j is X-empty with respect to e, and (ii) e is *sr*-anticipatable at the entry of j.

But this implies that e is *sr*-eliminatable from all nodes on the path $[s \ldots k]$ and k contains an *sr*-eliminatable occurrence of e (definition 3.2).

□

Theorem A.1 *$REDUND_i(e) \iff$ node i contains an sr-eliminatable occurrence of e.*

Proof :

$\mathrm{REDUND}_i = \mathrm{XANTLOC}_i \cdot \mathrm{SRPPIN}_i$ (eq. 9). Thus from lemma A.1(b), $\mathrm{REDUND}_i(e)$ $\iff$ (i) i contains a locally X-anticipatable occurrence of e, (ii) e is sr-anticipatable at the entry of node i, and (iii) e is sr-eliminatable from i.

Since (ii) is implied by (i) it follows that $\mathrm{REDUND}_i(e) \iff$conditions (i) and (iii) are satisfied by $i \iff i$ contains a sr-eliminatable occurrence of e (definition 3.2). □

Theorem A.2

(a) $INSERT_l(e) \iff$node l satisfies [DPS–1] with respect to expression e.

(b) $INSERT_SYNTH_{lm}(e) \iff$node lm satisfies [DPS–2] with respect to expression e.

Proof :

(a) From equation (10) we have :

$$\mathrm{INSERT}_l = \mathrm{SRPPOUT}_i \cdot (\neg\mathrm{SRPPIN}_l + \neg\mathrm{XTRANSP}_l) \cdot \neg\mathrm{SRAVOUT}_l$$

Then,

1. $\neg\mathrm{SRAVOUT}_l(e) \iff l$ satisfies DPS–1(a) with respect to e,
2. $\mathrm{SRPPOUT}_l(e) \iff l$ satisfies DPS–1(c) with respect to e (Lemma A.1(c)), and
3. $\neg\mathrm{SRPPIN}_l(e) + \neg\mathrm{XTRANSP}_l(e) \iff l$ satisfies DPS–1(b) with respect to e. A proof of this fact follows :

   $\neg\mathrm{XTRANSP}_l(e) \Rightarrow$ a computation of e placed at the exit of node l is not locally X-anticipatable and hence not sr-eliminatable from l (definition 3.2). Now consider $\neg\mathrm{SRPPIN}_l(e) \cdot \mathrm{XTRANSP}_l(e)$. From lemma A.1(c) $\mathrm{SRPPOUT}_l(e) \Rightarrow \mathrm{SRANTOUT}_l(e)$ which, with $\mathrm{XTRANSP}_l(e) \Rightarrow \mathrm{SRANTIN}_l(e)$. Hence, $\neg\mathrm{SRPPIN}_l(e) \Rightarrow e$ is not sr-eliminatable from node l (lemma A.1(b)).

   To prove the converse, if possible let $\mathrm{XTRANSP}_l(e) \cdot \mathrm{SRPPIN}_l(e)$. Then from lemma A.1(b) it follows that e is sr-eliminatable from node l — a contradiction.

(b) Similar to part (a).

□

Theorem A.3

(a) $\mathit{UINSERT\_EXIT}_k(e) \Longleftrightarrow$ *node* k *satisfies [DPS–3] with respect to expression* e*.*

(b) $\mathit{UINSERT\_ENTRY}_k(e) \Longleftrightarrow$ *node* k *satisfies [DPS–4] with respect to expression* e*.*

Proof :

(a) From equations (12), (13) and (14), $\mathrm{UINSERT\_EXIT}_k(e) \Longleftrightarrow k$ contains a downward exposed update definition for e and there exists a path $[k \ldots i]$ such that (i) $\mathrm{REDUND}_i(e)$, (ii) $\neg\mathrm{INSERT}_j(e)$ for all nodes j on the path $[k \ldots i)$, (iii) $\neg\mathrm{INSERT\_SYNTH}_{rs}(e)$ for every two consecutive nodes r and s on the path $[k \ldots i]$, and (iv) $\neg\mathrm{XCOMP}_r(e)$ for all nodes r on the path $(k \ldots i)$.

From theorem A.1, i contains an sr-eliminatable occurrence of e. Also from lemma 3.1 it follows that there exists a path $[l \ldots k \ldots i]$ in the optimized program such that $\mathrm{INSERT}_l(e)$ or $\mathrm{INSERT\_SYNTH}_{pl}(e)$ (where p is a predecessor of l) or $\mathrm{XCOMP}_l(e)$ and all nodes on the path $(l \ldots i)$ are X-empty with respect to e. Thus k satisfies DPS–3 with respect to e.

(b) $\mathrm{UINSERT\_ENTRY}_l = \mathrm{REDUND}_l \cdot \neg\mathrm{ANTLOC}_l$ (eq. (15)) and $\mathrm{REDUND}_l \Rightarrow \mathrm{XANTLOC}_l$ (eq. (9)). Hence, $\mathrm{UINSERT\_ENTRY}_l(e) \Longleftrightarrow \mathrm{REDUND}_l \cdot \mathrm{XANTLOC}_l \cdot \neg\mathrm{ANTLOC}_l \Longleftrightarrow l$ satisfies [DPS–4] with respect to e (theorem A.1).

□