A Fresh Look at PRE as a Maximum Flow Problem

Jingling Xue[1] and Jens Knoop[2]

[1] Programming Languages and Compilers Group, School of Computer Science and Engineering, University of New South Wales, Sydney, NSW 2052, Australia
[2] Technische Universität Wien, Institut für Computersprachen, Argentinierstraße 8, 1040 Wien, Austria

Abstract. We show that classic PRE is also a maximum flow problem, thereby revealing the missing link between classic and speculative PRE, and more importantly, establishing a common high-level conceptual basis for this important compiler optimisation. To demonstrate this, we formulate a new, simple unidirectional bit-vector algorithm for classic PRE based only on the well-known concepts of availability and anticipatability. Designed to find a unique minimum cut in a flow network derived from a CFG, which is proved simply but rigorously, our algorithm is simple and intuitive, and its optimality is self-evident. This conceptual simplicity also translates into efficiency, as validated by experiments.

1 Introduction

Partial redundancy elimination (PRE) is a compiler optimisation that eliminates computations that are redundant on some but not necessarily all paths in a program. As a result, PRE encompasses both global common subexpression elimination and loop-invariant code motion. Over the years, PRE has also been extended to perform other optimisations at the same time, including strength reduction [8, 12, 16, 18], global value numbering [3] and live-range determination [21]. For these reasons, PRE is regarded as one of the most important optimisations in optimising compilers.

As a code transformation, PRE eliminates a partially redundant computation at a point by inserting its copies on the paths that do not already compute it prior to the point, thereby making the partially redundant computation fully redundant. PRE problems come in two flavours: *classic PRE* and *speculative PRE*. Classic PRE, as described in the seminal work [22], inserts a computation at a point only if the point is *safe* (or *down-safe*) for the computation, i.e., only if the computation is fully anticipatable at the point. On the other hand, speculative PRE may insert a computation at a point even if the computation is partially but not necessarily fully anticipatable at the point. If the computation cannot cause an exception and if the execution frequencies of the flow edges in a CFG are available, speculative PRE may find transformations missed by classic PRE, thereby removing more redundancies in dynamic terms than classic PRE.

In the case of classic PRE, Knoop, Rüthing and Steffen invented an optimal unidirectional bit-vector formulation of the problem [17, 19]. This algorithm, known as Lazy Code Motion (LCM), was later recasted to operate in static single assignment (SSA) form [15]. Subsequently, a number of alternative formulations have been proposed [7, 8, 9, 23]. While LCM and other earlier algorithms [8, 9] find code insertion points by modelling the optimisation as a code motion transformation, the latter ones [7, 23] avoid this by identifying code insertion points directly. Apparently, a search for a conceptual basis upon which an optimal formulation of classic PRE can be both developed and understood more intuitively has been the driving force behind these research efforts. Up to now, however, this conceptual basis has been elusive. All existing algorithms are developed and reasoned about at the low level of individual program paths.

While classic PRE is profile-independent, speculative PRE is profile-guided. Given a weighted CFG, where the weights of the flow edges represent their execution frequencies, we have shown previously that speculative PRE is a maximum flow problem [26]. Finding an optimal transformation on a CFG amounts to finding a special minimum cut in a flow network derived from the CFG. Furthermore, different optimal transformations on a CFG may result if the weights of the flow edges in the CFG differ.

In this paper, we show for the first time that classic PRE is also a maximum flow problem. This is the key to the main contribution of our paper: to provide a uniform approach for classic and speculative PRE. The insight behind this finding lies in the following assumption made about classic PRE [17, 19]: all control flow edges are nondeterministic, or equivalently, have nonzero execution frequencies. We show that finding the optimal transformation for a CFG amounts to finding a unique minimum cut in a flow network derived from the CFG. Since all insertions in a CFG must be safe in classic PRE (as mentioned above), this unique minimum cut is invariant of the execution frequencies of the flow edges in the CFG. This establishes the connection and highlights the main difference between classic and speculative PRE. More importantly, our finding provides a common high-level conceptual basis upon which an optimal formulation of PRE can be more systematically and intuitively developed and proved. *Every PRE algorithm, if being optimal, must find the unique minimum cut on a flow network that is derived from a CFG.* As a result, tedious and non-intuitive reasoning that has been practised at the lower level of control flow paths is dispensed with.

Based on this insight, we have developed a new, simple algorithm for classic PRE. Our formulation, applicable to standard basic blocks, consists of solving four unidirectional bit-vector data-flow problems based only on the well-known concepts of availability and anticipatability. Designed to find a unique minimum cut in a flow network derived from a CFG, which is proved simply but rigorously, our data-flow equations reason positively about the global properties computed without using logical negations. Such a formulation is intuitive and its optimality self-evident. This conceptual simplicity also translates into efficiency, as demonstrated by our experimental results.

The rest of this paper is organised as follows. Section 2 gives the background information. Section 3 shows that classic PRE is a maximum flow problem. We do so constructively by giving an algorithm, MIN-PRE, that consists of solving three data-flow problems and invoking a min-cut algorithm to find a unique minimum cut in a flow network derived from a CFG. Section 4 compares and contrasts classic and speculative PRE when both are viewed as maximum flow problems. In Section 5, we derive from MIN-PRE a simple algorithm, called SIM-PRE, for classic PRE by solving four data-flow problems only. Section 6 discusses some experimental results. Our simple algorithm uses fewer bit-vector operations than three algorithms across 22 SPECcpu2000 benchmarks on two platforms. Section 7 reviews the related work and concludes.

2 Background

A control flow graph (CFG), $G = (N, E, W)$, is a weighted directed graph, where N is the set of basic blocks (or nodes), E the set of control flow edges and $W : N \cup E \mapsto \mathbb{N}$. Given a node or edge x, $W(x)$ represents its execution frequency (under an arbitrary input). In addition, *ENTRY* $\in N$ denotes its *entry block* and *EXIT* $\in N$ its *exit block*, which are both empty. Furthermore, every block is assumed to lie on some path from *ENTRY* to *EXIT*. Let $\mathit{pred}(G, n)$ be the set of all *immediate predecessors* of a block n in G and $\mathit{succ}(G, n)$ the set of all *immediate successors* of a block n in G.

Assumption 1. *For every $G = (N, E, W)$, we have the following tautology:*

$$\forall\, n \in N : \textstyle\sum_{m \in \mathit{pred}(G,n)} W(m, n) = \sum_{m \in \mathit{succ}(G,n)} W(n, m)$$

As in [17, 19], we consider a non-SSA intermediate representation, where each statement has the form $v = e$ such that v is a variable and e a single-operator expression. As is customary, we assume that local common subexpression elimination (LCSE) has already been applied to all basic blocks. Given an expression e, the following three local predicates associated with a block n are used in the normal manner. ANTLOC_n is true if e is locally anticipatable on entry to block n (i.e., block n contains an upwards exposed computation of e). COMP_n is true if e is locally available on exit from block n (i.e., block n contains a downwards exposed computation of e). TRANSP_n is true if block n does not contain any modification to e. PRE is a global optimisation. So only the upwards and downwards exposed computations of e, called the *PRE candidates*, will be considered. A block can contain at most two PRE candidate computations. It is important to be reminded that ANTLOC_n and COMP_n can both be true in block n, in which case, either a common PRE candidate of e is locally available and anticipatable simultaneously, implying that $\mathsf{TRANSP}_n = \mathit{true}$ or two distinct PRE candidates of e are locally available and anticipatable, respectively, implying that $\mathsf{TRANSP}_n = \mathit{false}$.

A PRE transformation for an expression is realised by replacing all redundant computations of the expression by a new temporary that is initialised correctly at suitable program points. We adopt the definition of PRE as used in LCM [19] except that we will make use of edge insertions as in [7, 23, 26] rather than node insertions; these insertions serve to make all the partially redundant computations fully redundant. Therefore, we do not have to split *critical edges*, i.e., the edges leading from nodes with more than one immediate successor to nodes with more than one immediate predecessor.

The fundamental assumption in classic PRE as stated clearly in LCM [17, 19] is that all control flows in a CFG are nondeterministic. Equivalently, we have:

Assumption 2. *Given $G = (N, E, W)$. In classic PRE,* $\forall\, x \in (N \cup E) : W(x) > 0$.

A directed graph $F = (V, A)$ is a *flow network* if it has two distinguished nodes, a *source* s and a *sink* t, in V and a nonnegative *capacity* (or *weight*) for each edge in A. Let S and $T = V - S$ be a partition of V such that $s \in S$ and $t \in T$. We denote by (S, T) the set of all (directed) edges with tail in S and head in T: $(S, T) = \{(n, m) \in A \mid n \in S, m \in T\}$. A *cut* separating s from t is any edge set $(C, \overline{C})$, where $s \in C$, $\overline{C} = V - C$ is the complement of C and $t \in \overline{C}$. The *capacity* of this cut is the sum of the capacities of all *cut edges* in the cut. A *minimum cut* is a cut separating s from t with minimum capacity. The *max-flow problem* consists of finding a flow of maximum value from the source s to the sink t. The max-flow min-cut theorem of [10] dictates that such a flow exists and has a value equal to the capacity of a minimum cut.

3 Classic PRE as a Maximum Flow Problem

In classic PRE, only safe insertions are used as discussed previously. Based on this safety constraint and Assumption 2, we show that classic PRE on a CFG is a maximum flow problem and a special minimum cut on a flow network derived from the CFG leads to the construction of the unique (lifetime) optimal transformation for the CFG — the uniqueness was known earlier in [17, 19]. These results provide a common high-level conceptual basis for developing, understanding and reasoning about PRE algorithms.

In Section 3.1, MIN-PRE is presented and illustrated by an example. In Section 3.2, we give an intuitive explanation why classic PRE is a maximum flow problem. In Section 3.3, we see that the optimality proof in this context is straightforward.

3.1 MIN-PRE

In classic PRE, a computation of an expression e is said to be *redundant* (partially or fully) if it can be eliminated by using safe code insertions of the form $t_e = e$, where t_e is a new temporary. A computation of e is said to *generate* some redundancies if it can cause another computation of e (both may be identical, as in a loop) to be redundant.

To shed the light on the nature of classic PRE on a CFG, we specify such a transformation for an expression e by using the following three sets (as in the GCC compiler):

DELETE gives the set of blocks where the upwards exposed computations of e are redundant (partially or fully). Every such computation will be replaced by a new temporary t_e. Note that a computation of e that is downwards but not also upwards exposed cannot be redundant (i.e., removable using safe code insertions only).

COPY gives the set of all *copy blocks* where the downwards exposed computations of e generate redundancies in the blocks given in DELETE but these computations themselves (when they are also upwards exposed) are not redundant. Such a computation will be replaced by t_e and preceded by a *copy* insertion of $t_e = e$. Note that a computation of e that is upwards but not also downwards exposed cannot generate any redundancies (i.e., cause other computations to be redundant).

INSERT gives the set of edges, called *insertion edges*, on which $t_e = e$ will be inserted, thereby making all partially redundant computations of e fully redundant.

This definition distinguishes clearly the different roles that the three different code modifications play in a PRE transformation. As we shall see shortly, DELETE and INSERT are so closely related that both can be built simultaneously. However, more information about redundancy-generating computations is needed in order to build COPY.

A transformation is *correct* if every use of t_e is identified with a definition of $t_e = e$ in every execution path. The total number of computations of e eliminated by a

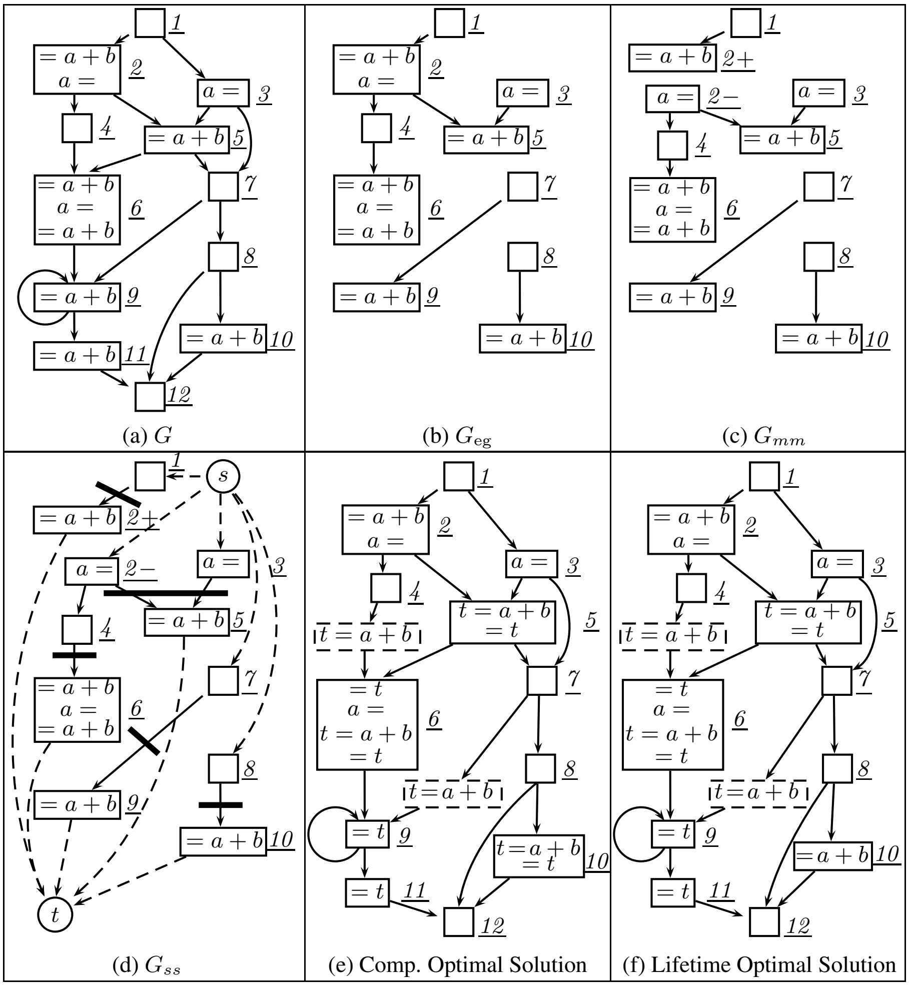


Fig. 1. A running example

transformation in $G = (N, E, W)$ is given by $\sum_{b \in \mathsf{DELETE}} W(b) - \sum_{e \in \mathsf{INSERT}} W(e)$. A transformation is *computationally optimal* if this term is maximised and is *lifetime optimal* (or *optimal* for short) if the live ranges of all definitions of t_e are also minimised.

Our running example is given in Figure 1. An optimal PRE algorithm will take as input the CFG shown in Figure 1(a) and produce as output the transformed CFG as shown in Figure 1(f). The optimal transformation for the example is specified by:

$$\begin{aligned} \mathsf{DELETE} &= \{6, 9, 11\} \\ \mathsf{COPY} &= \{5, 6\} \\ \mathsf{INSERT} &= \{(4,6), (7,9)\} \end{aligned} \tag{1}$$

Figure 2 gives a mincut-based algorithm, MIN-PRE, for classic PRE by modelling it as a maximum flow problem. The reader is invited to read the algorithm since it is made to be self-contained. Below we explain its steps and illustrate them by our example.

We start with a weighted CFG, $G = (N, E, W)$, where the weights of its blocks and flow edges are their execution frequencies. In the example given in Figure 1(a), we do not actually show the weights. As we shall see later, the optimal transformation in classic PRE is independent of the weights in a CFG. In Steps 1 and 2, we compute the standard global properties, availability and anticipatability, on G. Based on this information, in Step 3, we derive an important subgraph G_{eg} from G. G_{eg} contains every *essential edge* $(m, n) \in E$ such that $\mathsf{ESS}(m, n) = \overline{\mathsf{AVAILOUT}_m} \cdot \mathsf{ANTIN}_n$ holds and its two incident nodes $m, n \in N$. Figure 1(b) depicts this subgraph for the running example. By definition, the following two properties about G_{eg} are true.

Lemma 1. *Let n be a node in G such that $\mathsf{ANTLOC}_n = \mathit{true}$. If the upwards exposed computation in n is not fully redundant, then n is always included in G_{eg}.*

Note that n in the above lemma may also be contained in G_{eg} even if n is fully redundant, in which case, n must have at least one outgoing edge that is essential.

In Figures 1(a) and (b), we see that G_{eg} contains block 2, 5, 6, 9 and 10 but not 11.

Lemma 2. *For every computationally optimal transformation, its* INSERT *must be a subset of the edge set E_{eg} of G_{eg}.*

Proof. By Assumption 2, a transformation whose INSERT contains $(m, n) \notin E_{\text{eg}}$ such that $\mathsf{AVAILOUT}_m = \mathit{true}$ ($\mathsf{ANTIN}_n = \mathit{false}$) cannot be computationally optimal (safe). $\square$

Thus, G_{eg} is the fundamental subgraph of G where code insertions are to be made to make all the partially redundant computations (i.e., those that are removable by using safe code insertions only) in G fully redundant.

In Step 4, we obtain a multi-source, multi-sink flow network, G_{mm}, from G_{eg}. Figure 1(c) depicts this network for our example, where block 2 has been split (conceptually). Note that $N_{\text{eg}}^{\text{ss}} = \{2\}$. Intuitively, an insertion of $t = a + b$ that makes the upwards exposed computation $a + b$ in block 6 fully redundant must be made "below" block 2. Hence, the conceptual split. Note that the sources and sinks are: $S_{mm} = \{1, 2-, 3, 7, 8\}$ and $T_{mm} = \{2+, 5, 6, 9, 10\}$. By construction, $S_{mm} \cap T_{mm} = \emptyset$ holds. Finally, $\mathcal{N}$ ($\mathcal{E}$) relates the nodes (edges) in G_{mm} to those in G. We have $\mathcal{N}(2+) = \mathcal{N}(2-) = 2$ and $\mathcal{N}(n) = n$ for other nodes n. As a result, $\mathcal{E}(1, 2+) = (1, 2)$, $\mathcal{E}(2-, 4) = (2, 4)$, $\mathcal{E}(2-, 5) = (2, 5)$ and $\mathcal{E}(m, n) = (m, n)$ for other edges.

By construction, the following two lemmas about the structure of G_{mm} are immediate. Lemma 3 says that every sink in G_{mm} contains an upwards exposed computation (which cannot be fully redundant since at least one of its incoming edges is essential). The correctness of this lemma can be verified in Figure 1(c). Lemma 4 gives the key reason why classic PRE can be more efficiently solved by data-flow analysis only and also reveals its difference with speculative PRE (Section 4).

Lemma 3. *Let n be a node in G_{mm}. Then $n \in T_{mm}$ iff* $\mathsf{ANTLOC}_{\mathcal{N}(n)} = \mathit{true}$.

Proof. Follows from the construction of G_{eg} and the derivation of G_{mm} from G_{eg} by means of the conceptual split as conducted in Step 4 of MIN-PRE. $\square$

(1) Compute global availability on G:

$$\mathsf{AVAILIN}_n = \begin{cases} \mathit{false} & \text{if } n = \mathit{ENTRY} \\ \prod_{m \in \mathit{pred}(G,n)} \mathsf{AVAILOUT}_m & \text{otherwise} \end{cases}$$

$$\mathsf{AVAILOUT}_n = \mathsf{COMP}_n + \mathsf{AVAILIN}_n \cdot \mathsf{TRANSP}_n$$

(2) Compute global anticipatability on G:

$$\mathsf{ANTOUT}_n = \begin{cases} \mathit{false} & \text{if } n = \mathit{EXIT} \\ \prod_{m \in \mathit{succ}(G,n)} \mathsf{ANTIN}_m & \text{otherwise} \end{cases}$$

$$\mathsf{ANTIN}_n = \mathsf{ANTLOC}_n + \mathsf{ANTOUT}_n \cdot \mathsf{TRANSP}_n$$

(3) Define $G_{\text{eg}} = (N_{\text{eg}}, E_{\text{eg}}, W_{\text{eg}})$ as a subgraph of G:
$N_{\text{eg}} = \{n \in N \mid \exists\, m \in N : \mathsf{ESS}(m,n) \vee \exists\, m \in N : \mathsf{ESS}(n,m)\}$
$E_{\text{eg}} = \{(m,n) \in E \mid \mathsf{ESS}(m,n)\}$
where $\mathsf{ESS}(m,n) = \overline{\mathsf{AVAILOUT}_m} \cdot \mathsf{ANTIN}_n$ for all $(m,n) \in E$.

(4) Derive a multi-source, multi-sink network $G_{mm} = (N_{mm}, E_{mm}, W_{mm})$ from G_{eg} as follows. A *source* (*sink*) is a node without predecessors (successors). Let $N_{\text{eg}}^{\text{ss}} = \{n \in N_{\text{eg}} \mid \mathsf{ANTLOC}_n \wedge \overline{\mathsf{TRANSP}_n} \wedge \mathit{pred}(G_{\text{eg}}, n) \neq \emptyset \wedge \mathit{succ}(G_{\text{eg}}, n) \neq \emptyset\}$. For every such a *source-sink node* $n \in N_{\text{eg}}^{\text{ss}}$, containing instructions $I_1, \ldots, I_p$, such that I_k is the first modification to expression e, replace n by two new nodes $n+$ and $n-$, where $n+$ contains $I_1, \ldots, I_{k-1}$ and $n-$ contains $I_k, \ldots, I_p$, such that the incoming (outgoing) edges of n in G_{eg} are now directed into (out of) $n+$ ($n-$) and no edges exist between $n+$ and $n-$. (If I_k is of the form $h = e$ such that e is upwards exposed, and also modified by h, i.e., the LHS of I_k, then split conceptually $h = e$ into $h' = e; h = h'$ before splitting n.)
Let $S_{mm} = \{n \in N_{mm} \mid \mathit{pred}(G_{mm}, m) = \emptyset\}$ and $T_{mm} = \{n \in N_{mm} \mid \mathit{succ}(G_{mm}, m) = \emptyset\}$.
Let $\mathcal{N} : N_{mm} \mapsto N$ such that $\mathcal{N}(n+) = \mathcal{N}(n-) = \mathcal{N}(n) = n$.
Let $\mathcal{E} : E_{mm} \mapsto E$ such that $\mathcal{E}(m,n) = (\mathcal{N}(m), \mathcal{N}(n))$.

(5) Derive a single-source, single-sink flow network $G_{ss} = (N_{ss}, E_{ss}, W_{ss})$ from G_{mm} as follows. Introduce two new nodes, s and t, add an edge with weight ∞ from the *source* s to every node in S_{mm} and an edge with weight ∞ from every node in T_{mm} to the *sink* t.

(6) Find a unique minimum cut, $\mathcal{E}^{-1}(\mathcal{C}_\Lambda) = (\Lambda, \overline{\Lambda})$, in G_{ss}, as follows:

(a) Apply any min-cut algorithm to find a maximum flow f in G_{ss}.
(b) Let $G_{ss}^f = (N_{ss}, E_{ss}^f, W_{ss}^f)$ be the residual network induced by f [5], where
$E_{ss}^f = \{(u,v) \in E_{ss} \mid W_{ss}(u,v) - f(u,v) > 0\}$
$W_{ss}^f = E_{ss}^f \mapsto \mathbb{N}$, where $W_{ss}^f(u,v) = W_{ss}(u,v) - f(u,v)$
(c) Let $\overline{\Lambda} = \{n \in N_{ss} \mid \text{there is a path from } n \text{ to sink } t \text{ in } G_{ss}^f\}$ and $\Lambda = N_{ss} \setminus \overline{\Lambda}$.
(d) Let $\mathcal{C}_\Lambda = \mathcal{C}_\Lambda^{\text{ins}} \uplus \mathcal{C}_\Lambda^{\text{copy}}$, where $\mathcal{C}_\Lambda^{\text{copy}} = \{(m,n) \in \mathcal{C}_\Lambda \mid \forall\, p \in \mathit{pred}(G,n) : (p,n) \in \mathcal{C}_\Lambda\}$.

(7) Solve the "live range analysis for t_e" in G:

$$\mathsf{LIVEOUT}_n = \begin{cases} \mathit{false} & \text{if } n = \mathit{EXIT} \\ \sum_{m \in \mathit{succ}(G,n)} (\mathsf{LIVEIN}_m \cdot ((n,m) \notin \mathcal{C}_\Lambda)) & \text{otherwise} \end{cases}$$

$$\mathsf{LIVEIN}_n = \mathsf{ANTLOC}_n + \mathsf{LIVEOUT}_n \cdot \mathsf{TRANSP}_n$$

(8) Define the optimal transformation as follows:
$\mathsf{DELETE} = \{n \in N \mid \mathsf{ANTLOC}_n \wedge \mathsf{UE\text{-}REDUND}_n\}$
$\mathsf{COPY} = \{n \in N \mid \mathsf{COMP}_n \wedge \mathsf{LIVEOUT}_n \wedge (\overline{\mathsf{TRANSP}_n} \vee \overline{\mathsf{UE\text{-}REDUND}_n})\}$
$\mathsf{INSERT} = \mathcal{C}_\Lambda^{\text{ins}}$
where $\mathsf{UE\text{-}REDUND}_n = (\{(m,n) \in E \mid m \in \mathit{pred}(G,n)\} \not\subseteq \mathcal{C}_\Lambda^{\text{copy}})$ for all $n \in N$.

Fig. 2. A mincut-based algorithm, MIN-PRE, for classic PRE on $G = (N, E, W)$

Lemma 4. *For every $n \in N_{mm} \setminus (S_{mm} \cup T_{mm})$, the following statement is true:*

$$\sum_{m \in \mathit{pred}(G_{mm}, n)} W(m,n) \leqslant \sum_{m \in \mathit{succ}(G_{mm}, n)} W(n,m) \tag{2}$$

Proof. For every $n \in N_{mm} \setminus (S_{mm} \cup T_{mm})$, we must have $n \in G_{\text{eg}}$. It suffices to show that all outgoing edges of n in G are included in G_{mm}, i.e., $\forall\, m \in \mathit{succ}(G,n) : (n,m) \in N_{mm}$. By Lemma 3, $\mathsf{ANTLOC}_n = \mathit{false}$. Since $n \notin S_{mm}$, then there must exist an incoming edge (p,n) of n in G_{mm} such that $\mathsf{ESS}(p,n) = \mathit{true}$, i.e., $\mathsf{AVAILOUT}_p = \mathit{false}$ and $\mathsf{ANTIN}_n = \mathit{true}$. When $\mathsf{AVAILOUT}_p = \mathit{false}$, we have $\overline{\mathsf{AVAILIN}_n} = \mathit{false}$. Note that $\overline{\mathsf{ANTLOC}_n} \wedge \mathsf{ANTIN}_n \Longrightarrow \mathsf{TRANSP}_n$. Furthermore, $\overline{\mathsf{ANTLOC}_n} \wedge \mathsf{TRANSP}_n \Longrightarrow \overline{\mathsf{COMP}_n}$. Thus, $\mathsf{AVAILOUT}_n = \mathit{false}$ and $\mathsf{ANTOUT}_n = \mathit{true}$. When $\mathsf{ANTOUT}_n = \mathit{true}$, by definition, we have $\forall\, m \in \mathit{succ}(G,n) : \mathsf{ANTIN}_m = \mathit{true}$. Hence, $\forall\, m \in \mathit{succ}(G,n) : \mathsf{ESS}(n,m) = \overline{\mathsf{AVAILOUT}_n} \wedge \mathsf{ANTIN}_m = \mathit{true}$, implying that $\forall\, m \in \mathit{succ}(G,n) : (n,m) \in N_{mm}$. □

In Step 5, we obtain a single-source, single sink flow network G_{ss} from G_{mm} in the normal manner. In Step 6, we find a unique minimum cut $\mathcal{E}^{-1}(\mathcal{C}_\Lambda)$ on G_{ss} by applying the "Reverse" Labelling Procedure of [10], where $\overline{\Lambda}$ is the smallest possible (Lemma 8). Figure 1(d) depicts G_{ss} for our example, together with the following minimum cut:

$$\begin{aligned} \mathcal{C}_\Lambda &= \{(1,2),(2,5),(3,5),(4,6),(7,9),(8,10)\} \\ \mathcal{C}_\Lambda^{\text{ins}} &= \{(4,6),(7,9)\} \\ \mathcal{C}_\Lambda^{\text{copy}} &= \{(1,2),(2,5),(3,5),(8,10)\} \end{aligned} \tag{3}$$

Such a partition of $\mathcal{C}_\Lambda$ into $\mathcal{C}_\Lambda^{\text{ins}}$ and $\mathcal{C}_\Lambda^{\text{copy}}$ is significant due to the fundamentally different roles they play in defining DELETE, COPY and INSERT given in Step 8. According to their definitions in Step 6(d), $\mathcal{C}_\Lambda^{\text{ins}}$ ($\mathcal{C}_\Lambda^{\text{copy}}$) includes a cut edge (m,n) iff some but not all (all) incoming edges of n in the original CFG, G, are cut edges.

In order to define DELETE, we need to know if a computation is redundant or not. This is trivial for downwards but not also upwards exposed computations.

Lemma 5. *Let n be a node in G such that $\mathsf{COMP}_n = \mathit{true}$. If $\mathsf{TRANSP}_n = \mathit{false}$, then the downwards exposed computation in n is not redundant.*

To check if an upwards exposed computation is redundant or not, we apply Lemma 6, which makes use of the predicate $\mathsf{UE\text{-}REDUND}_n$ introduced at the end of Step 8.

Lemma 6. *Let n be a node in G_{mm}, where $n' = \mathcal{N}(n)$ is the corresponding node in G. Then $\mathsf{UE\text{-}REDUND}_{n'} = \mathit{false}$ iff the upwards exposed computation in n' is not redundant (i.e., not removable by using safe code insertions only).*

Proof. To prove "$\Longrightarrow$", we note that MIN-PRE finds the minimum cut $(\Lambda, \overline{\Lambda})$ by applying the "Reverse" Labelling Procedure of [10] to G_{ss}. Therefore, n must be a sink in G_{mm}, which implies $\mathsf{ANTLOC}_{n'} = \mathit{true}$ by Lemma 3. Let X be the set of all nodes in $G_{mm} \setminus (S_{mm} \cup T_{mm})$ lying on a path from a source in G_{mm} to n. By Lemma 4, $\mathsf{UE\text{-}REDUND}_{n'} = \mathit{false}$, i.e., all incoming edges of n' in G are included in $\mathcal{C}_\Lambda^{\text{copy}}$ iff $\forall\, p \in X : \sum_{m \in \mathit{pred}(G_{mm},p)} W(m,p) = \sum_{m \in \mathit{succ}(G_{mm},p)} W(p,m)$. To prove "$\Longleftarrow$", we know that n' must be contained in G_{eg} by Lemma 1. By Lemma 3, n is a sink node in G_{mm}. By Lemma 4, the upwards exposed computation in n' is not redundant iff $\forall\, p \in X : \sum_{m \in \mathsf{pred}(G_{mm},p)} W(m,p) = \sum_{m \in \mathsf{succ}(G_{mm},p)} W(p,m)$. A combination of the results proved so far concludes the proof. □

Looking at Step 8, we find that $\mathcal{C}_{\Lambda}^{\text{copy}}$ is sufficient for defining DELETE (Lemmas 5 and 6) while $\mathcal{C}_{\Lambda}^{\text{ins}}$ is sufficient for defining INSERT. However, Lemmas 5 and 6 do not tell us if a computation that is not redundant generates any redundancies or not. This means that some extra information is required in order to define COPY completely.

A naive solution is to copy at *all* blocks containing non-redundant computations:

$$\mathsf{COPY}^{\text{all}} = \{n \in N \mid \mathsf{COMP}_n \wedge (\overline{\mathsf{TRANSP}_n} \vee \overline{\mathsf{UE\text{-}REDUND}_n})\} \tag{4}$$

Then, $\mathsf{COPY}^{\text{all}}$, together with DELETE and INSERT given in Step 8, will yield a computationally optimal transformation (as implied by the proof of Theorem 1). In terms of LCM, this transformation corresponds to ALCM (Almost LCM) [17, 19].

For the running example, such a computationally optimal transformation is:

$$\begin{aligned} \mathsf{DELETE} &= \{6, 9, 11\} \\ \mathsf{COPY}^{\text{all}} &= \{5, 6, 10\} \\ \mathsf{INSERT} &= \{(4,6), (7,9)\} \end{aligned} \tag{5}$$

where DELETE and INSERT are the same as in (1). This results in the transformed code in Figure 1(e). However, the definition of t in block 10 is only used in that block. Such a copy operation should be avoided since the live range of t is unnecessarily introduced. The downwards exposed computations of this kind are known to be *isolated* [17, 19]. In Step 7, we solve a third data-flow problem so that $\mathsf{COPY} \subseteq \mathsf{COPY}^{\text{all}}$ is defined in Step 8 with all these isolated blocks being excluded. Note that a copy is required in a block n if it contains a downward exposed computation, which generates redundancies and is not upwards exposed (i.e., $\mathsf{COMP}_n \wedge \mathsf{LIVEOUT}_n \wedge \overline{\mathsf{TRANSP}_n}$ (Lemma 5)) or if it contains an upward exposed computation, which generates redundancies and is not redundant itself (i.e., $\mathsf{COMP}_n \wedge \mathsf{LIVEOUT}_n \wedge \overline{\mathsf{UE\text{-}REDUND}_n}$ (Lemma 6)).

This problem can be understood as one of solving the live variable analysis for temporary t_e on the transformed CFG realised by DELETE and INSERT and $\mathsf{COPY}^{\text{all}}$. By comparing $\mathsf{COPY}^{\text{all}}$ and COPY, we see that we include a block in COPY by also requiring t_e to be live on exit from that block. This guarantees that the downwards exposed computation of e in such a block must generate some redundancies.

Consider Figure 1(e), $\mathsf{LIVEOUT}_5 = \mathsf{LIVEOUT}_6 = \mathit{true}$ but $\mathsf{LIVEOUT}_{10} = \mathit{false}$. Hence, COPY includes only blocks 5 and 6. The final transformation is given in (1), which results in the optimally transformed code shown in Figure 1(f).

Remark. If we apply the (normal) Labelling Procedure of [10] in Step 6 of MIN-PRE, $(\Lambda, \overline{\Lambda})$ will be found such that $\overline{\Lambda}$ is the largest. The PRE transformation obtained using such a cut will correspond to the Busy Code Motion (BCM) as described in [17, 19].

3.2 (Full) Availability as the Single Commodity

Consider classic PRE carried out optimally on a CFG G for an expression e. All (partially or fully) redundant computations of e, which must be upwards exposed, as identified by DELETE, are *consumers* of the value of e. All downwards exposed computations of e that generate redundancies and are not redundant themselves, as identified by COPY, are *producers* of the value of e. Classic PRE can be modelled as a single-commodity maximum flow problem. The value of e (i.e., the commodity) is to be routed from the existing producers to the existing consumers under the condition that e must be (fully) available at the consumers. To achieve this full availability, new producers, as identified by INSERT, can be introduced in G, or precisely, G_{ss} under the safe constraint that only the existing consumers can receive the value of e. The cost of introducing these new producers (i.e., the number of computations of e incurred) is equal to the maximum flow on G_{ss}. In order to be computationally optimal, their placements are the cut edges of a minimum cut as implied in the proof of Theorem 1. In the optimal transformation, new producers must be installed according to the unique minimum cut found by applying essentially the "Reverse" Labelling Procedure of [10] to G_{ss}.

3.3 Optimality

First of all, we recall Lemma 10 from [14] on the structure of all minimum cuts.

Lemma 7. *If* $(A, \overline{A})$ *and* $(B, \overline{B})$ *are minimum cuts in an s-t flow network, then* $(A \cap B, \overline{A \cap B})$ *and* $(A \cup B, \overline{A \cup B})$ *are also minimum cuts in the network.*

This lemma implies immediately that a unique minimum cut $(C, \overline{C})$ exists such that $\overline{C}$ is the *smallest*, i.e., that $\overline{C} \subset \overline{C'}$ for every other minimum cut $(C', \overline{C'})$. Note that $\subset$ is strict. In addition, this lemma is valid independently of any maximum flow that one may use to enumerate all maximum cuts for the underlying network.

In fact, for the minimum cut $(\Lambda, \overline{\Lambda})$ found by MIN-PRE, $\overline{\Lambda}$ is the smallest.

Lemma 8. *Let* S_{cut} *be the set of minimum cuts in* $G_{ss} = (N_{ss}, E_{ss}, W_{ss})$. *Consider the minimum cut* $(\Lambda, \overline{\Lambda})$ *in* G_{ss} *found by MIN-PRE. Then:*

$$\overline{\Lambda} \subseteq \overline{C} \textit{ for all } (C, \overline{C}) \in S_{cut} \tag{6}$$

where the equality in $\subseteq$ *holds iff* $\overline{\Lambda} = \overline{C}$.

Proof. By Assumption 2, G_{ss} is an *s-t* flow network with positive edge capacities only. In Step 6 of MIN-PRE, we find the minimum cut $(\Lambda, \overline{\Lambda})$ by applying essentially the "Reverse" Labelling Procedure of [10]. Its construction ensures that (6) holds with respect to the maximum flow f used. Lemma 7 implies that this "smallest minimum cut" is independent of the maximum flow f. Hence, (6) is established. □

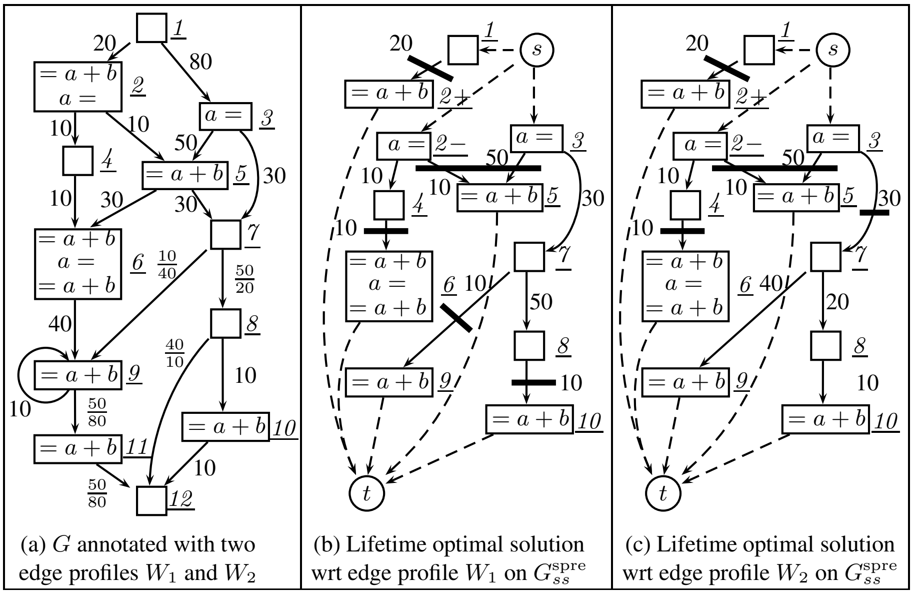
Theorem 1. *The transformation found by MIN-PRE is (lifetime) optimal and unique.*

Proof. Consider an expression e in $G = (N, E, W)$. Let LO denote the transformation found by MIN-PRE, which is represented by DELETE, INSERT and COPY. Let a lifetime optimal transformation be represented by DELETE_T, INSERT_T and COPY_T. By Lemma 6, $\mathsf{DELETE} = \mathsf{DELETE}_T$. By also applying Lemma 6 and noting that $n \in$ COPY iff it generates redundancies in DELETE, we must have $\mathsf{COPY} = \mathsf{COPY}_T$. Recall that $\mathcal{E}^{-1}(\mathcal{C}_\Lambda) = (\Lambda, \overline{\Lambda})$ is the minimum cut found by MIN-PRE in its Step 6, where $\mathcal{C}_\Lambda = \mathcal{C}_\Lambda^{\text{ins}} \uplus \mathcal{C}_\Lambda^{\text{copy}}$. By Lemma 2, INSERT_T must be drawn from the edges of G_{eg}. Clearly, $\mathcal{E}^{-1}(\mathcal{C}_\Lambda^{\text{copy}} \uplus \mathsf{INSERT}_T)$ must be a cut since T cannot be valid otherwise. Furthermore, $\mathcal{E}^{-1}(\mathcal{C}_\Lambda^{\text{copy}} \uplus \mathsf{INSERT}_T)$ must be a minimum cut. Otherwise, LO constructed using a minimum cut will cause fewer computations of e to be evaluated. Let $\mathcal{E}^{-1}(\mathcal{C}_\Lambda^{\text{copy}} \uplus \mathsf{INSERT}_T) = (\Lambda', \overline{\Lambda'})$. By Lemma 8, $\overline{\Lambda} \subseteq \overline{\Lambda'}$. Thus, the equality in $\overline{\Lambda} \subseteq \overline{\Lambda'}$ must hold. Otherwise, the live ranges of $t_e = e$ in LO will be better than those in T. Hence, $LO = T$ is lifetime optimal, which is unique since $(\Lambda, \overline{\Lambda})$ is. □

4 Classic PRE vs. Speculative PRE

In [26], we formulated speculative PRE as a maximum flow problem. This work shows that classic PRE is also a maximum flow problem. We recognise immediately that the fundamental difference between the two optimisations lies only in Step 2 of MIN-PRE. In the case of speculative PRE, we will compute partial anticipatability rather than full anticipatability. As a result, two seemingly different PRE problems are unified.

We compare and contrast classic and speculative PRE by using Figures 1 and 3. In Figure 3(a), our example CFG G is annotated with two different edge profiles. Figures 3(b) and 3(c) depict (identically) the flow network G_{ss}^{spre} obtained by applying MIN-PRE to G except that partial anticipatability is computed in its Step 2. Compared to G_{ss} in Figure 1(d) in classic PRE, G_{ss}^{spre} has two more edges: $(3, 7)$ and $(7, 8)$.



$= a + b$
$a =$
$= a + b$
s
t
1
2
2+
2−
3
4
5
6
7
8
9
10
11
12
20
80
10
50
30
$\frac{10}{40}$
$\frac{50}{20}$
$\frac{40}{10}$
$\frac{50}{80}$
40
(a) G annotated with two edge profiles W_1 and W_2
(b) Lifetime optimal solution wrt edge profile W_1 on G_{ss}^{spre}
(c) Lifetime optimal solution wrt edge profile W_2 on G_{ss}^{spre}

Fig. 3. Profile-sensitivity of speculative PRE. In (a), the CFG from Figure 1(a) is annotated with two different edge profiles. If an edge is labelled by x, $W_1(e) = W_2(e) = x$. If an edge label is $\frac{x}{y}$, $W_1(e) = x$ and $W_2(e) = y$. The optimal solutions for W_1 and W_2 are given in (b) and (c).

The difference between classic and speculative PRE translates into the structural difference between G_{mm} and G_{mm}^{spre}, from which G_{ss} and G_{ss}^{spre} are derived. Lemma 4 that is valid for G_{mm} in classic PRE is not valid for G_{mm}^{spre} in speculative PRE. For example, block 8 in G_{ss}^{spre} given in Figure 3(b) or 3(c) is one such a counterexample node. As a result, G_{ss}^{spre} is generally an arbitrary flow network, implying that speculative PRE needs to be solved optimally using a min-cut algorithm. In addition, speculative PRE is profile-sensitive. Different execution profiles can result in different optimal transformations as illustrated in Figures 3(b) and 3(c). In both cases, only the execution frequencies on edges $(7, 8)$ and $(7, 9)$ are different. Note that the solution shown in Figure 3(b) is the same as the one found in classic PRE (Figure 1(d)). In speculative PRE, the benefit of an optimal transformation depends on the accuracy of the profiling information used. More computations may be evaluated if the profiling information used is completely inaccurate. On the other hand, classic PRE is profile-independent and thus conservative. Never will more computations be evaluated in the transformed code. Due to Lemma 4, different profiles always result in the same optimal transformation, as implied in Theorem 1. The reader can verify that MIN-PRE will return exactly the same minimum cut in Figure 1(d) for the two execution profiles given in Figure 3(a).

5 SIM-PRE: A Simple Mincut-Motivated Algorithm

Due to the special structure of G_{mm}, and consequently, G_{ss}, as identified in Lemma 4, we can find the unique minimum cut $\mathcal{E}^{-1}(\mathcal{C}_\Lambda) = \mathcal{C}_\Lambda^{\text{ins}} \uplus \mathcal{C}_\Lambda^{\text{copy}} = (\Lambda, \overline{\Lambda})$ found in Step 6 of MIN-PRE by solving one data-flow problem. Based on MIN-PRE, we have developed a simple and efficient algorithm, called SIM-PRE and given in Figure 4, for

(1) Compute global availability on G:
(2) Compute global anticipatability on G:
(3) Compute global availability perceived to be done on the transformed CFG:

$$\mathcal{TG}\text{-AVAILIN}_n = \begin{cases} \mathit{false} & \text{if } n = \mathit{ENTRY} \\ \text{AVAILIN}_n + \text{ANTIN}_n \cdot \sum_{m \in \mathit{pred}(G,n)} \mathcal{TG}\text{-AVAILOUT}_m & \text{otherwise} \end{cases}$$

$$\mathcal{TG}\text{-AVAILOUT}_n = \text{COMP}_n + \mathcal{TG}\text{-AVAILIN}_n \cdot \text{TRANSP}_n$$

(4) Compute a restricted form of partial anticipatability on the transformed CFG:

$$\mathcal{TG}\text{-PANTOUT}_n = \begin{cases} \mathit{false} & \text{if } n = \mathit{EXIT} \\ \mathcal{TG}\text{-AVAILOUT}_n \cdot \sum_{m \in \mathit{succ}(G,n)} \mathcal{TG}\text{-PANTIN}_m & \text{otherwise} \end{cases}$$

$$\mathcal{TG}\text{-PANTIN}_n = \text{ANTLOC}_n + \mathcal{TG}\text{-PANTOUT}_n \cdot \text{TRANSP}_n$$

(5) Define the optimal transformation as follows:

$$\text{DELETE} = \{n \in N \mid \text{ANTLOC}_n \wedge \mathcal{TG}\text{-AVAILIN}_n\}$$
$$\text{COPY} = \{n \in N \mid \text{COMP}_n \wedge \mathcal{TG}\text{-PANTOUT}_n \wedge (\overline{\text{TRANSP}_n} \vee \overline{\mathcal{TG}\text{-AVAILIN}_n})\}$$
$$\text{INSERT} = \{(m, n) \in E \mid \overline{\mathcal{TG}\text{-AVAILOUT}_m} \wedge \mathcal{TG}\text{-AVAILIN}_n\}$$

Fig. 4. A mincut-motivated algorithm, SIM-PRE, for classic PRE on $G = (N, E, W)$

classic PRE by solving four data-flow problems. Steps 1 and 2 remain the same. In Step 3, we solve a data-flow problem in G but the problem can be understood as one of computing global availability on the optimally transformed graph, G_{opt}, of G. The two global properties, $\mathcal{TG}$-$\mathsf{AVAILIN}_n$ and $\mathcal{TG}$-$\mathsf{AVAILOUT}_n$, are defined for the entry and exit of every block n in G_{opt}. $\mathcal{TG}$-$\mathsf{AVAILOUT}_n$ is computed in the normal manner. In the case of $\mathcal{TG}$-$\mathsf{AVAILIN}_n$, an expression e is available on entry to block n in G_{opt} if it is already available in G. In addition, if e is available along some incoming edges of block n but not along some others (m, n) in G_{opt} and if e is (fully) anticipatable on entry to n in G, then $(m, n) \in \mathcal{C}_\Lambda^{\text{ins}}$ must be an insertion edge. After $t_e = e$ has been made on all these insertion edges, $\mathcal{TG}$-$\mathsf{AVAILIN}_n = \textit{true}$ will hold. Hence, we have:

$$\mathsf{UE\text{-}REDUND}_n = \mathcal{TG}\text{-}\mathsf{AVAILIN}_n \tag{7}$$

which leads directly to:

$$\begin{aligned} \mathcal{C}_\Lambda^{\text{ins}} &= \{(m,n) \in E \mid \overline{\mathcal{TG}\text{-}\mathsf{AVAILOUT}_m} \wedge \mathcal{TG}\text{-}\mathsf{AVAILIN}_n\} \\ \mathcal{C}_\Lambda^{\text{copy}} &= \textstyle\bigcup_{n \in N:\mathsf{ANTLOC}_n \wedge \overline{\mathsf{UE\text{-}REDUND}_n}} \{(m,n) \in E \mid m \in \textit{pred}(G,n)\} \end{aligned} \tag{8}$$

To define COPY, we do not use the "live variable analysis" given in Step 7 of MIN-PRE. Instead, we solve a different data-flow problem, which is simpler for two reasons. First, we do not need to compute the predicate $(m, n) \in \mathcal{C}_\Lambda$ on flow edges. Second, the meet operator will use fewer bit-vector operations than before. This problem can be understood as one of computing partial anticipatability for an expression e on the transformed graph G_{opt} but only at the points where e is available in G_{opt}. (Note that $\mathcal{TG}$-PANTIN_n, which is not used, can be true on entry to block n if $\mathsf{ANTLOC}_n = \textit{true}$.)

Theorem 2. *The transformation found by SIM-PRE is lifetime optimal.*

Proof. Let *LO* be the transformation found by MIN-PRE, which is represented by DELETE, INSERT and COPY and *SIM* the transformation found by SIM-PRE, which is represented by DELETE_{SIM}, INSERT_{SIM} and COPY_{SIM}. By Lemma 4, (7) and (8) hold. By Lemmas 5 and 6, $\mathsf{DELETE} = \mathsf{DELETE}_{SIM}$ and $\mathsf{INSERT} = \mathsf{INSERT}_{SIM}$. By definition, $\mathcal{TG}$-$\mathsf{PANTOUT}_n = \textit{true}$ iff t_e is live on exit from n. So $\mathcal{TG}$-$\mathsf{PANTOUT}_n = \mathsf{LIVEOUT}_n$. Thus, $\mathsf{COPY} = \mathsf{COPY}_{SIM}$. This means that *LO=SIM*. □

6 Experimental Results

We evaluate the efficiencies of SIM-PRE and three other algorithms (denoted by LCM-DS, LCM-DS+COPY and E-Path) in terms of the total number of bit-vector operations performed on benchmark programs. All algorithms are implemented in GCC 3.4.3 and invoked to operate at its RTL (Register Transfer Language). We have applied the four algorithms to all 22 C/C++/FORTRAN SPECcpu2000 benchmarks compiled on two different platforms: Intel Xeon and SUN UltraSPARC-III. Due to architectural differences, the RTL representations on two platforms are drastically different.

LCM-DS denotes the GCC's implementation of a variant of LCM that was described in [9]. This algorithm assumes that the result of an expression is always available in a distinct temporary. Therefore, COPY is not computed. Since this assumption is not

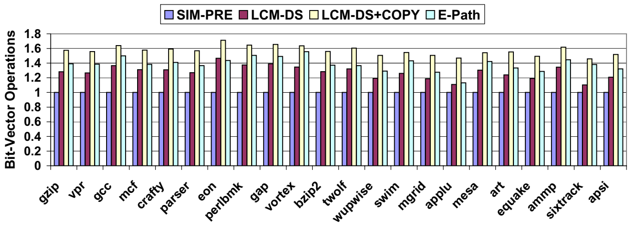


Fig. 5. A comparison of four algorithms on Xeon

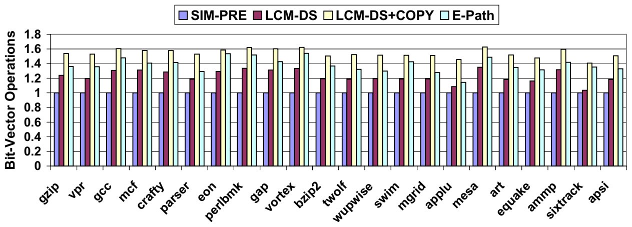


Fig. 6. A comparison of four algorithms on UltraSPARC-III

valid for RTL, GCC does a brute-force search on a CFG to compute COPY for each expression separately. There is no way to translate these graph traversal operations into equivalent bit-vector operations. Therefore, LCM-DS+COPY denotes the algorithm formed by combining LCM-DS and the 4th data-flow analysis used in SIM-PRE for computing COPY. E-Path is a recent new algorithm presented in [7].

SIM-PRE, LCM-DS+COPY and E-Path each solve four data-flow problems while LCM-DS solves only three (as discussed above). The first two problems, availability and anticipatability, are all the same. These algorithms differ only in how the remaining problem(s) are formulated. The efficiency of an algorithm is measured in terms of the number of bit-vector operations performed by all data-flow problems in an algorithm.

All algorithms are implemented using the bit-vector routines provided by GCC and operate on the same set of PRE candidate expressions used by GCC. A PRE candidate expression is always the RHS of an assignment, where the LHS is a virtual register. The RHS expressions that are constants or virtual registers are excluded (since no computations are involved). So are any expressions such as call expressions with side effects.

Figure 5 gives the (normalised) bit-vector operations consumed by four algorithms on Xeon. In LCM-DS (and LCM-DS+COPY), the data-flow equations for computing EARLIEST and LATER are expensive due to the excessive use of logical negations and somewhat complex equations employed. In E-Path, the equations used in the last two data-flow problems are more complex than those in SIM-PRE. In particular, the meet operators in Eps_in_i and SA_out_i are more expensive to evaluate. Figure 6 gives our experimental results on UltraSPARC-III. In both computer platforms, SIM-PRE requires fewer bit-vector operations than each of the other three algorithms. The key reason for SIM-PRE's efficiency is that the equations in solving its last two data-flow problems are simpler. Since these two problems are formulated to find a unique minimum cut for a CFG, we reason positively about the two global properties without using logical negations. So the number of bit-vector operations used are reduced.

7 Related Work and Conclusions

LCM [17, 19] and its extensions [8, 15] find code insertion points by modelling the optimisation as a code motion transformation as suggested in [6]. This approach is characterised by a few concepts, such as *earliest*, *latest* and *isolated*, that are not inherent in the PRE problem itself. Alternatively, some formulations of classic PRE [7, 23] avoid these concepts by identifying code insertion points directly. The driving force behind the development of these different formulations has probably been the insatiable desire to find a good conceptual basis upon which an optimal formulation of classic PRE can be easily developed, understood and reasoned about. However, in all existing formulations of classic PRE, data-flow equations are still cleverly but ad hocly designed. Their optimality is usually not obvious to their users and their proofs tedious and non-intuitive since the proofs have always been conducted at the low level of individual paths. This work provides a common high-level conceptual basis upon which an optimal formulation of PRE can be developed and proved. All optimal algorithms must find one way or another the unique minimum cut on a flow network G_{ss} derived from a CFG.

Classic PRE has been extended to perform other important optimisations, including strength reduction [8, 12, 16, 18], global value numbering [3], live-range determination [21], code size reduction [24], redundant load/store elimination [21] and data speculation [20]. Its scope has also been extended by means of code restructuring [2].

The earliest papers on speculative PRE can be found in [11, 13]. There are three computationally optimal algorithms for speculative PRE [1, 4, 25]. Later we also developed a lifetime optimal algorithm [26]. This work shows that both seemingly different problems are inherently related and can be unified under a common conceptual basis.

PRE is an important optimisation in optimising compilers and also serves as a classic classroom example for iterative and worklist data-flow analysis. The results and insights presented in this work are expected to be valuable in both settings.

References

1. R. Bodik. *Path-Sensitive Value-Flow Optimizations of Programs.* PhD thesis, University of Pittsburgh, 1999.
2. R. Bodik, R. Gupta, and M. L. Soffa. Complete removal of redundant computations. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 1–14, 1998.
3. P. Briggs and K. D. Cooper. Effective partial redundancy elimination. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 159–170, 1994.
4. Q. Cai and J. Xue. Optimal and efficient speculation-based partial redundancy elimination. In *1st IEEE/ACM International Symposium on Code Generation and Optimization*, pages 91–104, 2003.
5. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* Cambridge, Mass.: MIT Press, 1990.

6. D. M. Dhamdhere. A fast algorithm for code movement optimisation. *SIGPLAN Not.*, 23(10):172–180, 1988.
7. D. M. Dhamdhere. E-path_pre: partial redundancy elimination made easy. *SIGPLAN Not.*, 37(8):53–65, 2002.
8. V. M. Dhaneshwar and D. M. Dhamdhere. Strength reduction of large expressions. *Journal of Programming Languages*, 3:95–120, 1995.
9. K.-H. Drechsler and M. P. Stadel. A variation on Knoop, Rüthing, and Steffen's lazy code motion. *SIGPLAN Notices*, 28(5):29–38, 1993.
10. L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.
11. R. Gupta, D. A. Berson, and J. Z. Fang. Path profile guided partial redundancy elimination using speculation. In *Proceedings of the 1998 International Conference on Computer Languages*, pages 230–239, 1998.
12. M. Hailperin. Cost-optimal code motion. *ACM Transactions on Programming Languages and Systems*, 20(6):1297 – 1322, 1998.
13. R. Horspool and H. Ho. Partial redundancy elimination driven by a cost-benefit analysis. In *8th Israeli Conference on Computer System and Software Engineering*, pages 111–118, 1997.
14. T. C. Hu. *Integer Programming and Network Flows*. Addison-Wesley, 1970.
15. R. Kennedy, S. Chan, S.-M. Liu, R. Lo, and P. Tu. Partial redundancy elimination in SSA form. *ACM Transactions on Programming Languages and Systems*, 21(3):627–676, 1999.
16. R. Kennedy, F. C. Chow, P. Dahl, S.-M. Liu, R. Lo, and M. Streich. Strength reduction via SSAPRE. In *Proceedings of the 7th International Conference on Compiler Construction*, pages 144–158, London, UK, 1998. Springer-Verlag.
17. J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 224–234, 1992.
18. J. Knoop, O. Rüthing, and B. Steffen. Lazy strength reduction. *Journal of Programming Languages*, 1(1):71 – 91, 1993.
19. J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: Theory and practice. *ACM Trans. Program. Lang. Syst.*, 16(4):1117–1155, 1994.
20. J. Lin, T. Chen, W.-C. Hsu, P.-C. Yew, R. D.-C. Ju, T.-F. Ngai, and S. Chan. A compiler framework for speculative analysis and optimizations. In *Proceedings of the ACM SIGPLAN '03 Conference on Programming Language Design and Implementation*, pages 289–299, 2003.
21. R. Lo, F. Chow, R. Kennedy, S.-M. Liu, and P. Tu. Register promotion by sparse partial redundancy elimination of loads and stores. *SIGPLAN Not.*, 33(5):26–37, 1998.
22. E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Commun. ACM*, 22(2):96–103, 1979.
23. V. K. Paleri, Y. N. Srikant, and P. Shankar. A simple algorithm for partial redundancy elimination. *SIGPLAN Not.*, 33(12):35–43, 1998.
24. O. Rüthing, J. Knoop, and B. Steffen. Sparse code motion. In *Conference Record of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (*Boston, Massachusetts*), pages 170 – 183. ACM, New York, 2000.
25. B. Scholz, R. N. Horspool, and J. Knoop. Optimizing for space and time usage with speculative partial redundancy elimination. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 221–230, 2004.
26. J. Xue and Q. Cai. Profile-guided partial redundancy elimination using control speculation: a lifetime optimal algorithm and an experimental evaluation. Technical Report UNSW–CSE–TR–0420, School of Computer Science and Engigeering, University of New South Wales, Jul. 2004.