Complete Removal of Redundant Expressions

Rastislav Bodík Rajiv Gupta Mary Lou Soffa

Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
{bodik,gupta,soffa}@cs.pitt.edu

Abstract

Partial redundancy elimination (PRE), the most important component of global optimizers, generalizes the removal of common subexpressions and loop-invariant computations. Because existing PRE implementations are based on *code motion*, they fail to completely remove the redundancies. In fact, we observed that 73% of loop-invariant statements cannot be eliminated from loops by code motion alone. In dynamic terms, traditional PRE eliminates only half of redundancies that are strictly partial. To achieve a *complete* PRE, control flow *restructuring* must be applied. However, the resulting code duplication may cause code size explosion.

This paper focuses on achieving a complete PRE while incurring an acceptable code growth. First, we present an algorithm for *complete* removal of partial redundancies, based on the *integration* of code motion and control flow restructuring. In contrast to existing complete techniques, we resort to restructuring merely to remove obstacles to code motion, rather than to carry out the actual optimization.

Guiding the optimization with a profile enables additional code growth reduction through selecting those duplications whose cost is justified by sufficient execution-time gains. The paper develops two methods for determining the optimization benefit of restructuring a program region, one based on path-profiles and the other on data-flow frequency analysis. Furthermore, the abstraction underlying the new PRE algorithm enables a simple formulation of speculative code motion guaranteed to have positive dynamic improvements. Finally, we show how to balance the three transformations (code motion, restructuring, and speculation) to achieve a near-complete PRE with very little code growth.

We also present algorithms for efficiently computing dynamic benefits. In particular, using an elimination-style data-flow framework, we derive a demand-driven frequency analyzer whose cost can be controlled by permitting a bounded degree of conservative imprecision in the solution.

Keywords: partial redundancy elimination, control flow restructuring, speculative execution, demand-driven frequency data-flow analysis, profile-guided optimization.

1 Introduction

Partial redundancy elimination (PRE) is a widely used and effective optimization aimed at removing program statements that are redundant due to recomputing previously produced values [26]. PRE is attractive because by targeting statements that are redundant only along some execution paths, it subsumes and generalizes two important value-reuse techniques: global common subexpression elimination and loop-invariant code motion. Consequently, PRE serves as a unified value-reuse optimizer.

Most PRE algorithms employ *code motion* [11, 12, 14, 15, 16, 17, 24, 26], a program transformation that reorders instructions without changing the shape of the control flow graph. Unfortunately, code-motion alone fails to remove routine redundancies. In practice, one half of computations that are strictly partially redundant (*not* redundant along some paths) are left unoptimized due to code-motion obstacles. In theory, even the optimal code-motion algorithm [24] breaks down on loop invariants in while-loops, unless supported by explicit do-until conversion. Recently, Steffen demonstrated that control flow restructuring can remove from the program all redundant computations, including conditional branches [30]. While his property-oriented expansion algorithm *(Poe)* is complete, it causes unnecessary code duplication.

As the first step towards a complete PRE with affordable code growth, this paper presents a new PRE algorithm based on the *integration* of code motion and control flow restructuring, which allows a complete removal of redundant expressions while minimizing code duplication. No prior work systematically treated combining the two transformations. We control code duplication by restricting its scope to a *code-motion preventing* (CMP) region, which localizes adverse effects of control flow on the desired value reuse. Whereas the Poe algorithm applied to expression elimination (denoted *PoePRE*) uses restructuring to carry out the entire transformation, we apply the more economical code-motion transformation to its full extent, resorting to restructuring merely to enable the necessary code motion. The resulting code growth is provably not greater than that of PoePRE; on **spec95**, we found it to be three times smaller.

Second, to answer the overriding question of how complete a feasible PRE algorithm is allowed to be, we move from theory to practice by considering profile information. Using the dynamic amount of eliminated computations as the measure of optimization benefit, we develop a profile-guided PRE algorithm that limits the code growth cost

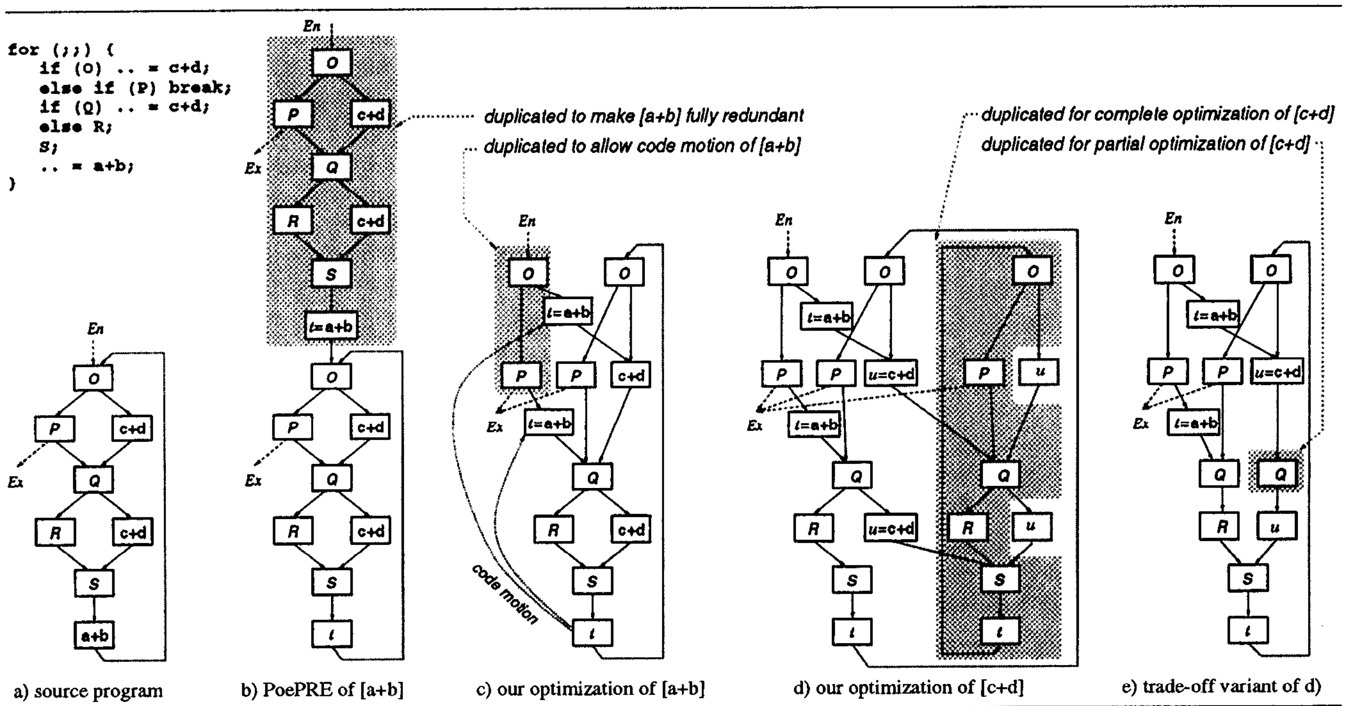


Figure 1: Complete PRE through integration of code motion and control flow restructuring.

by sacrificing those value-reuse opportunities that are infrequent but require significant duplication. Third, we describe how and when speculative code motion can be used instead of restructuring, and how to guarantee that speculative PRE is profitable. Finally, we demonstrate that a near-complete PRE with very little code growth can be achieved by integrating the three PRE transformations: pure code motion, restructuring, and speculative code motion.

All algorithms in this paper rely in a specific way on the notion of the CMP region which is used to reduce both code duplication and the program analysis cost. Thus, we make the PRE optimization more usable not only by increasing its *effectiveness* (power) through cost-sensitive restructuring, but also by improving its *efficiency* (implementation). We develop compile-time techniques for determining the impact of restructuring a program region on the dynamic amount of eliminated computations. The run-time benefit corresponds to the cumulative execution frequency of control flow paths that will permit value reuse after the restructuring. We describe how this benefit can be obtained either using edge profiles, path-profiles [7], or through data-flow *frequency analysis* [27].

As another contribution, we reduce the cost of frequency analysis by presenting a frequency analyzer derived from a new demand-driven data-flow analysis framework. Based on interval analysis, the framework enables formulation of analyzers whose time complexity is independent of the lattice size. This is a requirement of frequency analysis whose lattice is of infinite-height. Due to this requirement, existing demand frameworks are unable to produce a frequency analyzer [18, 22, 29]. Furthermore, we introduce the notion of *approximate* data-flow frequency information, which conservatively underestimates the meet-over-all-paths solution, keeping the imprecision within a given degree. Approximation permits the analyzer to avoid exploring program paths guaranteed to provide insignificant contribution (frequency-wise) to the overall solution. Besides PRE, the demand-driven approximate frequency analysis is applicable in interprocedural branch correlation analysis [10] and dynamic optimizations [5].

Let us illustrate our PRE algorithms on the loop in Figure 1(a). Assume no statement in the loop defines variables a, b, c, or d. Although the computations $[a+b]$ and $[c+d]$ are loop-invariant, removing them from the loop with code motion is not possible. Consider first the optimization of $[a+b]$. This computation cannot be moved out of the loop because it would be executed on the path En, O, P, Ex, which does not execute $[a+b]$ in the original program. Because this could slow down the program and create spurious exceptions, PRE disallows such *unsafe* code motion [24].

The desired optimization is only possible if the CFG is restructured. The PoePRE algorithm [30] would produce the program in Figure 1(b), which was created by duplicating each node on which the value of $[a+b]$ was available only on a subset of incoming paths. While $[a+b]$ is fully optimized, the scope of restructuring is unnecessarily large. Our complete optimization *(ComPRE)* produces the program in Figure 1(c), where code duplication is applied merely to enable the necessary code motion. In this example, to move $[a+b]$ out of the loop, it is sufficient to separate out the offending path En, O, P, Ex which is encapsulated in the CMP region highlighted in the figure. As no opportunities for value reuse remain, the resulting optimization of $[a+b]$ is complete. Because restructuring may generate irreducible programs, as in Figure 1(c), we also present a restructuring transformation that maintains reducibility.

Hoisting the loop invariant $[a+b]$ out of the loop was prevented by the shape of control flow. Our experiments show that the problem of removing loop invariant code (LI) has not been sufficiently solved: a complete LI is prevented for 73% of loop-invariant expressions. In some cases, a simple transformation may help. For example, $[a+b]$ (but not $[c+d]$) can be optimized by peeling off one loop iteration and performing the traditional LI [1], producing the program Figure 1(b). In while-loops, LI can often be enabled with more economical do-until conversion. The example presented does not allow this transformation because the loop exit does not post-dominate the loop entry. In effect, our restructuring PRE is always able to perform the smallest necessary do-until conversion for an arbitrary loop.

Next, we optimize the computation $[c+d]$ in Figure 1(c). Our optimization performs a complete PRE of $[c+d]$ by duplicating the shaded CMP region and subsequently performing the code motion (Figure 1(d)). The resulting program may cause too much code growth, depending on the sizes of duplicated basic blocks. Assume the size of block S outweighs the run-time gains of eliminating the upper $[c+d]$. In such a case, we select a smaller set of nodes to duplicate, as shown in Figure 1(e). When only block Q is duplicated, the optimization is no longer complete; however, the optimization cost measured as code growth is justified with the corresponding run-time gain. In Section 3.2, speculative code motion is used to further reduce code duplication.

In summary, this paper makes the following contributions:

- We present an approach for integrating two widely used code transformation techniques, *code motion* and *code restructuring*. The result is an algorithm for PRE that is *complete* (i.e., it exploits all opportunities for value reuse) and minimizes the code growth necessary to achieve the code motion.
- We show that restricting the algorithm to code motion produces the traditional code-motion PRE [17, 24].
- Profile-guided techniques for limiting the code growth through integration of selective duplication and speculative code motion are developed.
- We develop a demand-driven frequency analyzer based on a new *elimination* data-flow analysis framework.
- The notion of *approximate* data-flow information is defined and used to improve analyzer efficiency.
- Our experiments compare the power of code-motion PRE, speculative PRE, and complete PRE.

Section 2 presents the complete PRE algorithm. Section 3 describes profile-guided versions of the algorithm and Section 4 presents the experiments. Section 5 develops the demand-driven frequency analyzer. The paper concludes with a discussion of related work.

2 Complete PRE

In this section, we develop an algorithm for complete removal of partial redundancies (ComPRE) based on the integration of code motion and control flow restructuring. Code motion is the primary transformation behind ComPRE. To reduce code growth, restructuring is used only to enable hoisting through regions that prevent the necessary code motion. The smallest set of motion-blocking nodes is identified by solving the problems of availability and anticipability on an expressive lattice. We also show that when control flow restructuring is disabled, ComPRE becomes equivalent to the optimal code-motion PRE algorithm [24].

An expression is *partially redundant* if its value is computed on some incoming control flow path by a previous expression. Code-motion PRE eliminates the redundancy by hoisting the redundant computation along all paths until it reaches an edge where the reused value is available along either *all* paths or *no* paths. In the former case, the computation is removed; in the latter, it is inserted to make the original computation fully redundant. Unfortunately, code motion may be blocked before such edges are reached. Nodes that prevent the desired code motion are characterized by the following set of conditions:

1. hoisting of expression e across node n is *necessary* when
   a) an optimization candidate follows n: there is a computation of e downstream from n on some path, and
   b) there is a value-reuse opportunity for e at node n: a computation of e precedes n on some path.
2. hoisting of e across n is *disabled* when
   c) any path going through n does not compute e in the source program: such path would be impaired by the computation of e.

All three conditions are characterizable via solutions to the data-flow problems of anticipability and availability, which are defined as follows.

Definition 1 Let p be any path from the **start** node to a node n. The expression e is *available* at n along p iff e is computed on p without subsequent redefinition of its operands. Let r be any path from n to the **end** node. The expression e is *anticipated* at n along r iff e is computed on r before any of its operands are defined. The availability of e at the entry of n w.r.t. the incoming paths is defined as:

$$AVAIL_{in}[n,e] = \begin{cases} \mathit{Must} & \\ \mathit{No} & \text{if } e \text{ is available along} \\ \mathit{May} & \end{cases} \begin{matrix} \mathit{all} \\ \mathit{no} \\ \mathit{some} \end{matrix} \text{ paths.}$$

Anticipability ($ANTIC$) is defined analogously.

Given this refined value-reuse definition, code motion is necessary when a) and b) defined above hold mutually. Hence,

$$\begin{aligned} Necessary[n,e] &= ANTIC_{in}[n,e] \neq \mathit{No} \wedge \\ & \quad AVAIL_{in}[n,e] \neq \mathit{No}. \end{aligned}$$

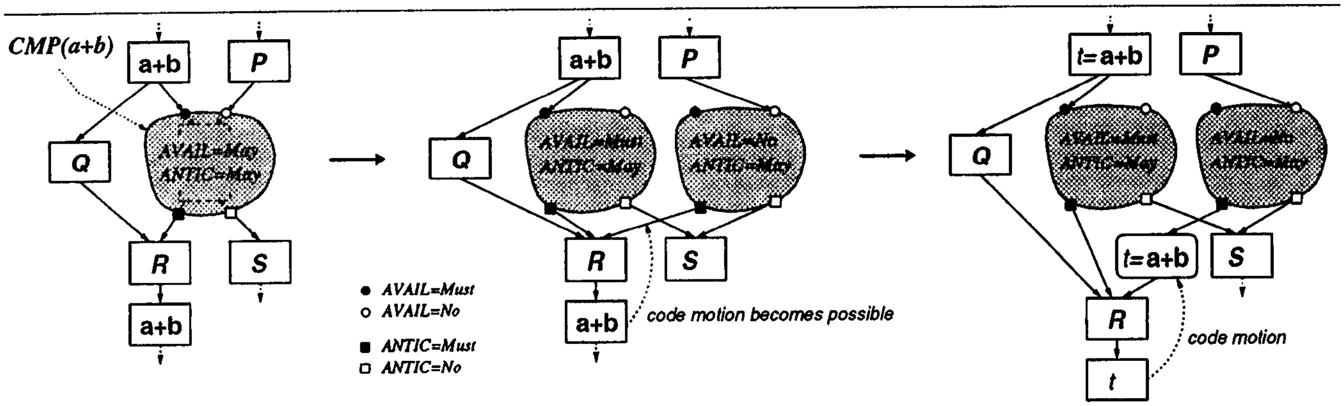
Code motion is disabled when the condition c) holds:

$$\begin{aligned} Disabled[n,e] &= ANTIC_{in}[n,e] \neq \mathit{Must} \wedge \\ & \quad AVAIL_{in}[n,e] \neq \mathit{Must}. \end{aligned}$$

A node n prevents the necessary code motion for e when the motion is necessary but disabled at the same time. By way of conjunction, we get the code motion-preventing condition:

$$\begin{aligned} Prevented[n,e] &= Necessary[n,e] \wedge Disabled[n,e] \\ &= ANTIC_{in}[n,e] = \mathit{May} \wedge \\ & \quad AVAIL_{in}[n,e] = \mathit{May} \end{aligned}$$

The predicate *Prevented* characterizes the smallest set of nodes that must be removed for code motion to be enabled.



a) code motion prevented by CMP region b) CMP region diluted via code duplication c) complete PRE of [a+b]

Figure 2: Removing obstacles to code motion via restructuring.

Definition 2 *Code Motion Preventing* region, denoted $CMP[e]$, is the set of nodes that prevent hoisting of a computation e: $CMP[e] = \{n \mid ANTIC_{in}[n,e] = \mathit{May} \wedge AVAIL_{in}[n,e] = \mathit{May}\}$.

To enable code motion, ComPRE removes obstacles presented by the CMP region by duplicating the entire region, as illustrated in Figure 2. The central idea is to factor the *May*-availability that holds in the entire region into *Must*- and *No*-availability, to hold respectively in each region copy. An alternative view is that we separate within the region the paths with *Must*- and *No*-availability. To achieve this, we can observe that a) no region entry edge is *May*-available, and b) the solution of availability within the region depends solely on solutions at entry edges (the expression is neither computed nor killed within the region). Hence, the desired factoring can be carried out by attaching to each region copy the subset of either *Must* or *No* entry edges, as shown in Figure 2(c).

After the CMP is duplicated, the condition *Prevented* is false on each node, enabling code motion. The ComPRE algorithm, shown in Figure 3, has the following three steps:

1. *Compute anticipability and availability.* The problems use the lattice $L = (\{\top, \mathit{Must}, \mathit{No}, \mathit{May}\}, \wedge)$. Note that the flow functions are distributive under the least common element operator $\wedge$, which is defined using the partial order $\sqsubseteq$ shown below. Distributivity property implies that data-flow facts are not approximated at control flow merge points. Intuitively, this is because L is the powerset lattice of $\{\mathit{No}, \mathit{Must}\}$, which are the only facts that may hold along an individual path.

   The partial order $\sqsubseteq$: $$\begin{array}{ccc} & T & \\ \mathit{No} & & \mathit{Must} \\ & \mathit{May} & \end{array}$$

2. *Remove CMP regions via control flow restructuring.* Given an expression e, the CMP region is identified by examining the data-flow solutions locally at each node. Line 2 in Figure 3 duplicates each CMP node and line 3 adjusts the control flow edges, so that the new copy of the region hosts the *Must* solution. Restructuring necessitates updating data-flow solutions within the CMP region (lines 4–12). While the $ANTIC$ solution is not altered, the previously computed $AVAIL$ solution is invalidated because some paths flowing into the region were eliminated when region entry edges were disconnected. For the expression e, $AVAIL$ becomes either *Must* or *No* in the entire region. For other expressions, the solution may become (conservatively) imprecise. In other words, splitting a *May* path into *Must*/*No* paths for e might have also split a *May* path for some other expression. Therefore, line 6 resets the initial guess and lines 10–12 recompute the solution within the CMP.

3. *Optimize the program.* The code motion transformation is carried out by replacing each original computation e with a temporary variable t_e. The temporary is initialized with a computation inserted into each *No*-available edge that sinks either into a *May*/*Must*-availability path or into an original computation. The insertion edge must also be *Must*-anticipated, to verify hoisting of the original computation to the edge.

Theorem 1 (Completeness). ComPRE is optimal in that it minimizes the number of computations on each path.

Proof. First, each original computation is replaced with a temporary. Second, no computation is inserted where its value is available along any incoming path. Hence, no additional computations can be removed. □

Within the domain of the Morel and Renviose code-motion transformation, where PRE is accomplished by hoisting optimization candidates (but not other statements) [26], ComPRE achieves minimum code growth.[1] This follows from the fact that after CMP restructuring, no program node can be removed or merged with some other node without destroying any value reuse, as shown by the following observations. Prior to Step 2, each node n may belong to CMP regions of multiple offending expressions. Duplication of n during restructuring can be viewed as partitioning of control flow paths going through n: each resulting copy of n is a path partition that does not contain both a *Must*- and a *No*-available path, for any offending expression. The

[1] Outside this domain, further code growth reduction is possible by moving instructions out of the CMP before its duplication.

Step 1: Data-flow analysis: anticipability, availability.

- **Input:** control flow graph $G = (N, E, \textbf{start}, \textbf{end})$, each node contains a single assignment $x := e$,
- $Comp(n, e)$: node n computes an expression e,
- $Transp(n, e)$: node n does not assign any variable in e,
- *boundary conditions:* for each expression e $ANTIC_{out}[\textbf{end}, e] := AVAIL_{in}[\textbf{start}, e] := No$,
- *initial guess:* set all vectors to $\top^S$, where S is the number of candidate expressions. Solve iteratively.

$$ANTIC_{in}[n,e] := \begin{cases} Must & \text{if } Comp(n,e), \\ No & \text{if } \neg Comp(n,e) \wedge \neg Transp(n,e), \\ ANTIC_{out}[n,e] & \text{otherwise.} \end{cases}$$

$$ANTIC_{out}[n,e] := \bigwedge_{m \in succ(n)} ANTIC_{in}[m,e]$$

$$AVAIL_{in}[n,e] := \bigwedge_{m \in pred(n)} AVAIL_{out}[m,e]$$

$$AVAIL_{out}[n,e] := f_n^e(AVAIL_{in}[n,e])$$

$$f_n^e(x) = \begin{cases} Must & \text{if } Comp(n,e) \wedge Transp(n,e), \\ No & \text{if } \neg Transp(n,e), \\ x & \text{otherwise.} \end{cases}$$

Step 2: Remove CMP regions: control flow restructuring.

- modify G so that no CMP nodes exists, for any expression e.

```
1  for each expression e do
       duplicate all CMP[e] nodes to create a copy of the CMP.
       n_Must is a copy of node n hosting AVAIL = Must.
2      N := N ∪ {n_Must | n ∈ CMP[e]}
       attach new nodes to perform the restructuring
3      E := (E ∪ {(n_Must, v) | (n, v) ∈ E ∧ v ∉ CMP[e]} ∪
           {(u, n_Must) | (u, n) ∈ E ∧ AVAIL_out[u, e] = Must} ∪
           {(n_Must, v_Must) | (n, v) ∈ E}) \
           {(u, n) ∈ E | u ∉ CMP[e] ∧ AVAIL_out[u, e] = Must}
       update data-flow solutions within CMP and its copy
4      for each node n ∈ CMP[e] do
5          ANTIC_in[n_Must] := ANTIC_in[n]
           ANTIC_out[n_Must] := ANTIC_out[n]
6          AVAIL_in[n_Must] := AVAIL_in[n] := ⊤^S
           AVAIL_out[n_Must] := AVAIL_out[n] := ⊤^S
7          AVAIL_in[n_Must, e] := AVAIL_out[n_Must, e] := Must
8          AVAIL_in[n, e] := AVAIL_out[n, e] := No
9      end for
       reanalyze availability inside both CMP copies
10     for each expression e' not yet processed do
11         re-compute AVAIL[n, e'], AVAIL[n_Must, e'], n ∈ CMP[e]
12     end for
13 end for
```

Step 3: Optimize: code motion.

$$Insert[(n,m),e] \Leftrightarrow ANTIC_{in}[m,e] = Must \wedge AVAIL_{out}[n,e] = No \wedge (AVAIL_{in}[m,e] = May \vee Comp(m,e))$$

$$Replace[n,e] \Leftrightarrow Comp(n,e)$$

Figure 3: ComPRE: the algorithm for complete PRE.

following properties of Step 2 can be verified: 1) the number of path partitions (node copies) created at a given node is independent of the order in which expressions are considered (in line 1), 2) each node copy is reachable from the **start** node, and 3) for any two copies of n there is an expression e such that remerging the two copies and their incoming paths will prevent code motion of e across the resulting node.

To compare ComPRE with a restructuring-only PRE, we consider PoePRE, a version of Steffen's complete algorithm [30] that includes minimization of duplicated nodes but is restricted in that only expressions are eliminated (as is the case in ComPRE). Elimination is carried out using a temporary, as in Step 3.

Theorem 2 ComPRE does not create more new nodes than PoePRE.

Proof outline. The proof is based on showing that the PoePRE-optimized program after minimization has no less nodes than the same program after CMP restructuring. It can be shown that, given an original node n, for any two copies of n created by CMP restructuring, there are two distinct copies of n created by PoePRE such that the minimization cannot merge them without destroying some value reuse opportunity.

In fact, PoePRE can be expressed as a form of ComPRE on a (non-strictly) larger region: for each computation e, PoePRE duplicates $\{n | ANTIC_{in}[n,e] \in \{Must, May\} \wedge AVAIL_{in}[n,e] = May\}$, which is a superset of $CMP[e]$.

Algorithm complexity. Data-flow analysis in Step 1 and in lines 10–12 requires $O(NS)$ steps, where N is the flow graph size and S the number of expressions. The restructuring in Step 2, however, may cause N to grow exponentially, as each node may need to be split for multiple expressions. Because in practice a constant-factor code-growth budget is likely to be defined, the asymptotic program size will not change. Therefore, the running time of Step 2, which dominates the entire algorithm, is $O(NS^2)$.

2.1 Optimal Code-Motion PRE

Besides supporting a complete PRE, the notion of the CMP region also facilitates an efficient formulation of code-motion PRE, called *CM-PRE*. In this section, we show that our complete algorithm can be naturally constrained by prohibiting the restructuring, and that such modification results in the same optimization as the optimal motion-only PRE [17, 24].

In comparison to ComPRE, the constrained CM-PRE algorithm bypasses the CMP removal; the last step (transformation) is unchanged (Figure 3). The first step (data-flow analysis) is modified with the goal to prevent hoisting across a node n when such motion would subsequently be blocked by a CMP region on *each* path flowing into node n. First, anticipability is computed as in ComPRE. Second, availability is modified to include detection of CMP nodes. When a CMP node is found, instead of propagating forward *May*-availability, the solution is adjusted to *No*. Such adjustment masks those value reuse opportunities that cannot be exploited without restructuring. The result of masking is that code motion is prevented from entering paths that cross a CMP region (see predicate *Insert* in Step 3 of Figure 3).

The modified flow function for the $AVAIL$ problem follows. The third line detects a CMP node. *No*-availability is now extended to mean that the value might be available along some path but value reuse is blocked by a CMP region along that path.

$$f_n^e(x) = \begin{cases} \mathsf{Must} & \text{if } Comp(n,e) \wedge Transp(n,e), \\ \mathsf{No} & \text{if } \neg Transp(n,e), \\ \mathsf{No} & \text{if } x = \mathsf{May} \wedge ANTIC_{in}[n,e] = \mathsf{May}, \\ x & \text{otherwise.} \end{cases}$$

Given a maximal fixed point solution to redefined $AVAIL$, CM-PRE performs the unchanged transformation phase (Figure 3, Step 3). It is easy to show that the resulting optimization is complete under the immutable shape of the control flow graph. The proof is analogous to that of Theorem 1: all original computations are removed and no computation has been inserted where an optimization opportunity *not blocked* by a CMP exists.

Besides exploiting all opportunities, a PRE algorithm should guarantee that the live ranges of inserted temporary variables are minimal, in order to moderate the register pressure. The live range is minimal when the insertion point specified by the predicate *Insert* cannot be delayed, that is, moved further in the direction of control flow.

Theorem 3 (Shortest live ranges). Given the CMP-restructured (or original) control flow graph, ComPRE (CM-PRE) is optimal in that it minimizes the live range lengths of inserted temporary variables.

Proof. An initialization point *Insert* cannot be delayed either because it would become partially redundant, destroying completeness, or because its temporary variable is used in the immediate successor. □

Existing PRE algorithms find the live-range optimal placement in two stages. First, computations are hoisted as high as possible, maximizing the removal of redundancies. Later, the placement is corrected through the computation of *delayability* [24]. Our formulation specifies the optimal placement directly, as we never hoist into paths where a blocking CMP will be subsequently encountered.

However, note that after the above redefinition, f_n^e is no longer monotone: given $ANTIC_{in}[n,e] = \mathsf{May}$, $x_1 = \mathsf{May}$, $x_2 = \mathsf{Must}$, we have $x_1 \sqsubseteq x_2$ but $f_n^e(x_1) = \mathsf{No} \not\sqsubseteq f_n^e(x_2) = \mathsf{Must}$. Although a direct approach to solving such system of equations may produce conservatively imprecise solution, the desired maximal fixed point is easily obtained using bit-vector GEN/KILL operations as follows.

First, compute $ANTIC$ as in Figure 3. Second, solve the well-known availability property, denoted $\mathbf{AV}_{all}$, which holds when the expression is computed along all incoming paths: $\mathbf{AV}_{all} \Leftrightarrow AVAIL = \mathsf{Must}$. Finally, we compute $\mathbf{AV}_{some}$ which characterizes *some*-paths availability and also encapsulates CMP detection: $\mathbf{AV}_{some} \Leftrightarrow AVAIL \neq \mathsf{No}$. The pair of solutions $(\mathbf{AV}_{all}, \mathbf{AV}_{some})$ can be directly mapped to the desired solution of $AVAIL$. The GEN and KILL sets [1] for the $\mathbf{AV}_{some}$ problem are given below. The value of the initial guess is *false*, the meet operator is the bit-wise **or**.

$$\begin{aligned} \mathbf{GEN} &= Comp \wedge Transp \\ \mathbf{KILL} &= \neg Transp \vee (AVAIL \neq \mathsf{Must} \wedge ANTIC \neq \mathsf{Must}) \\ & \neg Transp \vee (\neg \mathbf{AV}_{all} \wedge ANTIC \neq \mathsf{Must}) \end{aligned}$$

The condition $(AVAIL \neq \mathsf{Must} \wedge ANTIC \neq \mathsf{Must})$ detects the CMP node. While it is less strict than that in Definition 2, it is equivalent for our purpose, as it is safe to kill

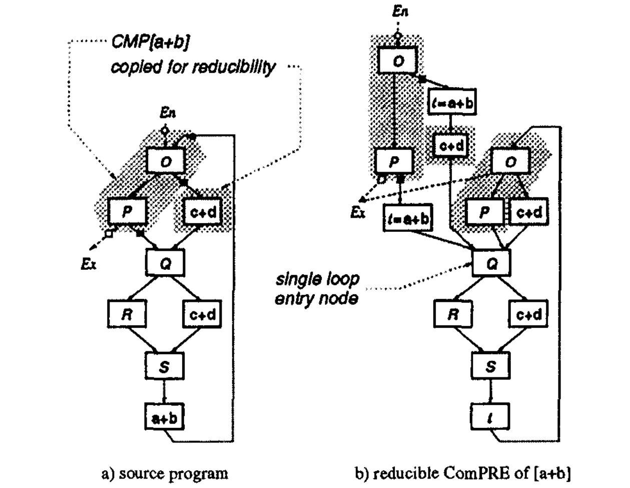


Figure 4: Reducible restructuring. (See Figure 1(c))

when there is no reuse ($AVAIL = \mathsf{No}$) or when there is no hoisting ($ANTIC = \mathsf{No}$). The less strict condition is beneficial because computing and testing Must requires one bit per expression, while two bits are required for May. Consequently, we can substitute $ANTIC \neq \mathsf{Must}$ with $\neg\mathbf{AN}_{all}$, where $\mathbf{AN}_{all}$ is defined analogously to $\mathbf{AV}_{all}$. As a result, we obtain the same implementation complexity as the algorithms in [17, 24]: three data-flow problems must be solved, each requiring one bit of solution per expression.

In conclusion, the CMP region is a convenient abstraction for terminating hoisting when it would unnecessarily extend the live ranges. It also provides an intuitive way of explaining the shortest-live-range solution without applying the corrective step based on delayability [24]. Furthermore, the CMP-based, motion-only solution can be implemented as efficiently as existing shortest-live-range algorithms.

2.2 Reducible Restructuring

Duplicating a CMP region may destroy reducibility of the control flow graph. In Figure 1(c), for example, ComPRE resulted in a loop with two distinct entry nodes. Even though PoePRE preserves reducibility on the same loop (Figure 1(b)), like other restructuring-based optimizations [4, 10, 30], it is also plagued by introducing irreducibility. One way to deal with the problem is to perform all optimizations that presuppose single-entry loops prior to PRE. However, many algorithms for scheduling (which should follow PRE) rely on reducibility.

After ComPRE, a reducible graph can be obtained with additional code duplication. An effective algorithm for normalizing irreducible programs is given in [23]. To suppress an unnecessary invocation of the algorithm, we can employ a simple test of whether irreducibility may be created after a region duplication. The test is based upon examining only the CMP entry and exit edges, rather than the entire program. Assuming we start from a reducible graph, restructuring will make a loop L irreducible only if multiple CMP exit edges sink into L, and at least one region entry is outside L (i.e., is not dominated by L's header node). If such a region is duplicated, target nodes of region exit edges may become the (multiple) loop entry nodes. Consider the loop in Figure 4(a). Two of the three exits of $CMP[a+b]$ fall into the loop. After restructuring, they will become loop entries, as shown in Figure 1(c).

Rather than applying a global algorithm like [23], a straightforward approach to make the affected loop reducible is to peel off a part of its body. The goal is to extend the replication scope so that the region exits sink onto a single loop node, which will then become the new loop entry. Such a node is the closest common postdominator (within the loop) of all the offending region exits and the original loop entry. Figure 4(a) highlights node **c+d** whose duplication after CMP restructuring will restore reducibility of the loop. The postdominator of the offending exits is node Q, which becomes the new loop header.

3 Profile-Guided PRE

While the CMP region is the smallest set of nodes whose duplication enables the desired code motion, its size is often prohibitive in practice. In this section, relying on the profile to estimate optimization benefit, complete PRE is made more practical by avoiding unprofitable code replication.

First, we extend ComPRE by *inhibiting restructuring* in response to code duplication *cost* and the expected dynamic *benefit*. The resulting profile-guided algorithm duplicates a CMP region only when the incurred code growth is justified by a corresponding run-time gain from eliminating the redundancies. Second, the notion of the CMP region is combined with profiling to formulate a *speculative* code-motion PRE that is guaranteed to have a positive dynamic effect, despite impairing certain paths. The third algorithm integrates both restructuring and speculation and selects a profitable subgraph of the CMP for each. While profitably balancing the cost and benefit under a given profile is NP-hard, the empirically small number of hot program paths promises an efficient algorithm [4, 19]. Finally, to support profile guiding, we show how an estimate of the run-time gain thwarted by a CMP region can be obtained using edge profiles, frequency analysis [27], or path profiles [7].

3.1 Selective Restructuring

We model the profitability of duplicating a CMP region R with a cost-benefit threshold predicate $T(R)$, which holds if the region optimization benefit exceeds a constant multiple of the region size. Our metric of benefit is the dynamic amount of computations whose elimination will be enabled after R is duplicated, denoted $Rem(R)$. That is, $T(R) = Rem(R) > c\ .size(R)$. When $T(R) = true$ for each region R, the algorithm is equivalent to the complete ComPRE. When $T(R) = false$ for each region, the algorithm reduces to the code-motion-only CM-PRE. Obviously, predicate T determines only a sub-optimal tradeoff between exploiting PRE opportunities and limiting the code growth. In particular, it does not explicitly consider the instruction cache size and the increase in register pressure due to introduced temporary variables. We have chosen this form of T in order to avoid modeling complex interactions among compiler stages. In the implementation, T is supplemented with a code growth budget (for example, in [6], code is allowed to grow by about 20%).

First, we present an algorithm for computing the optimization benefit $Rem(R)$. The method is based on the fact

```
Step 1: compute anticipability and availability.  (unchanged)
Step 2: Partial restructuring: remove profitable CMP regions.
for each computation e do
    for each disconnected subregion R_i of CMP[e] do
        build the largest connected subregion
        select a node from R and
        collect all connected CMP nodes
        determine optimization benefit Rem(R_i)
        carry out frequency analysis of AVAIL on R_i
        if profitable, duplicate (lines 2-12 of Fig. 3)
        if T(R_i) then duplicate R_i
    end for
end for
recompute the AVAIL solution, using f_n^e from Section 2.1
Step 3: Optimize: code motion.  (unchanged)
```

Figure 5: PgPRE: profile-guided version of ComPRE.

that the CMP scope *localizes* the entire benefit thwarted by the region: to compute the benefit, it suffices to examine only the paths within the region. Consider an expression e and its CMP region $R = CMP[e]$. For each region *exit* edge $a = (n, m)$ (i.e., $n \in CMP[e], m \notin CMP[e]$), the value of $ANTIC_{in}[m, e]$ is either *Must* or *No*, otherwise m would be in $CMP[e]$. Let $Exit_{Must}(R)$ be the set of the *Must* exit edges. The dynamic benefit is derived from the observation that each time such an edge is executed, any outgoing path contains *exactly* one computation of e that can be eliminated if: i) R is duplicated *and* ii) the value of e is available at the exit edge. Let $ex(a)$ be the execution frequency of edge a and $p(AVAIL_{out}[n,e] = Must)$ the probability that the value e is available when n is executed. After the region is duplicated, the expected benefit connected with the exit edge a is $ex(a).p(AVAIL_{out}[n,e] = Must)$, which corresponds to the number of computations removed on all paths starting at a. The benefit of duplicating the region R is thus the sum of all exit edge benefits

$$Rem(R) = \sum_{a=(n,m)\in Exits_{Must}(R)} ex(a).p(AVAIL_{out}[n,e] = Must).$$

The probability p is computed from an edge profile using *frequency analysis* [27]. In the frequency domain, the *probability* of each data-flow fact occurring, rather than the fact's mere boolean meet-over-all-paths existence, is computed by incorporating the execution probabilities of control flow edges into the data-flow system. Because the frequency analyzer cannot exploit bit-vector parallelism, but instead computes data-flow solutions on floating point numbers, it is desirable to reduce the cost of calculating the probabilities. The CMP region lends itself to effectively restricting the scope of the program that needs to be analyzed. Because all region entry edges are either *Must*- or *No*-available, the probability of e being available on these edges are 1 and 0, respectively. Therefore, the probability p at any exit edge can only be influenced by the paths within the region. As a result, it is sufficient to perform the frequency analysis for expression e on $CMP[e]$, using entry edges as a precise boundary condition for the CMP data-flow equation system. In Section 5 we reduce the cost of frequency analysis through a demand-driven approach.

The algorithm (PgPRE) that duplicates only profitable CMP regions is given in Figure 5. It is structured as its complete counterpart, ComPRE: after data-flow analysis, we proceed to eliminate CMP regions, separately for each expression. While in ComPRE it was sufficient to treat all nodes from a single CMP together, selective duplication benefits from dividing the CMP into disconnected subregions, if any exist. Intuitively, hoisting of a particular expression may be prevented by multiple groups of nodes, each in a different part of the procedure. Therefore, line 3 groups nodes from a connected subregion and frequency analysis determines the benefit of the group (line 4). After all profitable regions are eliminated, the motion-blocking effect of CMP regions remaining in the program must be captured. All that is needed is to apply the CM-PRE algorithm from Section 2.1 on the improved control flow graph. Blocked hoisting is avoided by recomputing availability (line 8) using the re-defined flow function f_n^e from Section 2.1, which asserts *No*-availability whenever a CMP is detected.

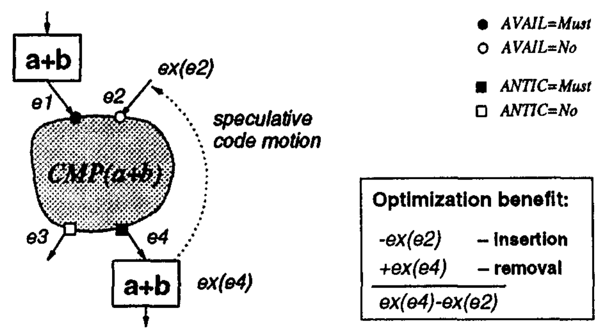


Figure 6: Speculative code-motion PRE.

3.2 Speculative Code-Motion PRE

In *code-motion* PRE, hoisting of a computation e is blocked whenever e would need to be placed on a control flow path p that does not compute e in the original program. Such *speculative* code motion is prevented because executing e along path p could a) raise spurious exceptions in e (e.g., overflow, wrong address), and b) outweigh the dynamic benefit of removing the original computation of e. The former restriction can be relaxed for instruction that cannot except, leading to *safe* speculation. New processor generations will support *control-speculative* instructions which will suppress raising the exception until the generated value is eventually used, allowing *unsafe* speculation [25]. The latter problem is solved in [20], where an aggressive code-motion PRE navigated by path profiles is developed. The goal is to allow speculative hoisting, but only into such paths on which dynamic impairment would not outweigh the benefit of eliminating the computation from its original position.

Next, we utilize the CMP region to determine i) the profitability of speculative code motion and ii) the positions of speculative insertion points that minimize live ranges of temporary variables. Figure 6 illustrates the principle of speculative PRE [20]. Instead of duplicating the CMP region, we hoist the expression into all *No*-available entry edges. This makes all exits fully available, enabling complete removal of original computations along the *Must* exits. In our example, $[a+b]$ is moved into the *No*-available region entry edge e_2. This hoisting is speculative because $[a+b]$ is now executed on each path going through e_2 and e_3, which previously did not contain the expression. The benefit is computed as follows. The dynamic amount of computations is decreased by the execution frequency $ex(e_4)$ of the *Must*-anticipable exit edge (following which a computation was removed), and increased by the frequency $ex(e_2)$ of the *No*-available entry edge (into which the computation was inserted). Since speculation is always associated with a CMP region, we are able to obtain a simple (but precise) *profitability* test: speculative PRE of an expression is profitable if the total execution frequency of *Must*-anticipable exit edges exceeds that of *No*-availaible entry edges. Note that the benefit is calculated locally by examining only entry/exit edges, and not the paths within the region, which was necessary in selective restructuring. Hence, the speculative benefit is independent from branch correlation and edge profiles are as precise as path profiles in the case of speculative-motion PRE. As far as temporary live ranges are concerned, insertion into entry edges results in a shortest-live-range solution, and Theorem 3 still holds.

3.3 Partial Restructuring, Partial Speculation

In Section 3.1, edge profiles and frequency analysis were used to estimate the benefit *Rem* of duplicating a region. An alternative is to use *path profiles* [3, 7], which are convenient for establishing cost-benefit optimization trade-offs [4, 19, 20]. To arrive at the value of the region benefit with a path profile, it is sufficient to sum the frequencies of *Must-Must* paths, which are paths that cross any region entry edge that is *Must*-available and any exit edge that is *Must*-anticipated. These are precisely the paths along which value reuse exists but is blocked by the region. While there is an exponential number of profiled acyclic paths, only 5.4% of procedures execute more than 50 distinct paths in **spec95** [19]. This number drops to 1.3% when low-frequency paths accounting for 5% of total frequency are removed. Since we can afford to approximate by disregarding these infrequent paths, summing individual path frequencies constitutes a feasible algorithm for many CMP regions. Furthermore, because they encapsulate branch correlation, path profiles compute the benefit more precisely than frequency analysis based on correlation-insensitive edge profiles.

Moreover, the notion of individual CMP paths leads to a better profile-guided PRE algorithm. Considering the CMP region as an indivisible duplication unit is overly conservative. While it may not be profitable to restructure the entire region, the region may contain a few paths *Must-Must* paths that are frequently executed and are inexpensive to duplicate. Our goal is to find the largest subset (frequency-wise) of region paths that together pass the threshold test $T(R)$. Similarly, speculative hoisting into *all* entry edges may fail the profitability test. Instead, we seek to find a subset of entry edges that maximizes the speculative benefit. In this section, we show how *partial* restructuring and speculation are carried out and combined.

Partial speculation selects for speculative insertion only a subset I of the *No* region entries. The selection of entries influences which subset R of region exits will be able to exploit value reuse. R consists of all *Must* exits that will become *Must*-available due to the insertions in I. The rationale behind treating entries separately is that some entries may enable little value reuse, hence they should not be speculated. Note that *No* entry edges are the only points where speculative insertion needs to be considered: insertions inside the region would be partially redundant; insertions outside the region would extend the live-ranges. Partial speculation is *optimal* if the difference of total frequencies of R and I is maximal (but non-negative). Although this problem is NP-

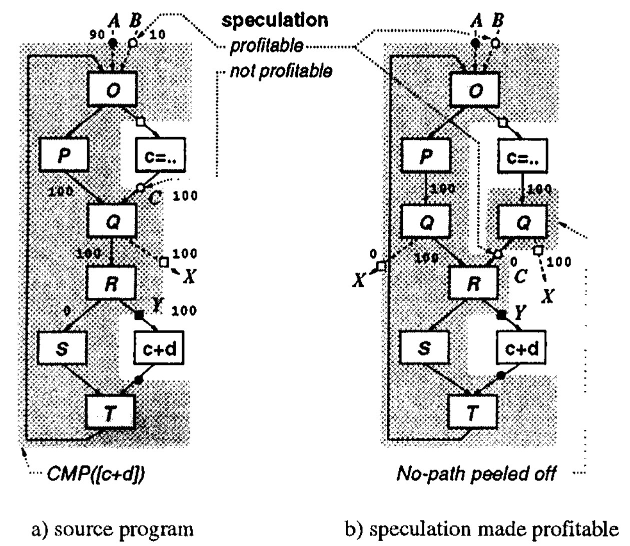


Figure 7: Integrating speculation and restructuring.

hard, the small number of entry edges observed in practice (typically less than 10) makes the problem tractable. An interesting observation is that to determine optimal partial speculation, a) edge profiles are not inferior to path profiles and b) frequency analysis is not required. Therefore, to exploit the power of path profiles, partial restructuring, rather than (speculative) code motion alone, must be used. This becomes more intuitive once we realize that without control flow restructuring, one is restricted to consider only an individual edge (but not a path) for expression insertion and removal. To compare the CMP-based partial speculation with the speculative PRE in [20], we show how to efficiently compute the benefit by defining the CMP region and how to apply edge profiles with the same precision as path profiles. In acyclic code, we achieve the same precision; in cyclic code, we are more precise in the presence of loop-carried reuse.

The task of *partial restructuring* is to localize a subgraph of the CMP that has a small size but contains many hot *Must-Must* paths. By duplicating only such a subregion, we are effectively peeling off only hot paths with few instructions. In Figure 1(e), only the (presumably hot) path through the node Q was separated. Again, the problem of finding an *optimal* subregion, one whose benefit is maximized but passes the $T(R)$ predicate and is smaller than a constant budget, is NP-hard. However, the empirically very small number of hot paths promises an efficient exhaustive-search algorithm.

Integrating partial speculation and restructuring offers additional opportunities for improving the cost-benefit ratio. We are no longer restricted to peeling off hot *Must-Must* paths and/or selecting *No*-entries for speculation. When the high frequency of a *No* entry prevents speculation, we can peel off a hot *No*-available path emanating from the entry, thereby reducing entry edge frequency and allowing the speculation, at the cost of some code duplication. Figure 7(a) shows an example program annotated with an edge profile. Because peeling hot *Must-Must* paths from the highlighted $CMP([c+d])$ would duplicate all blocks except S, we try speculation. To eliminate the redundancy at the CMP exit edge Y with frequency $ex(Y) = 100$, a computation must be inserted into *No*-entries B and C. While B is low-frequency (10), C is not (100), hence the speculation is disadvantageous, as $ex(Y) = 100 < ex(B) + ex(C) = 10 + 100$. Now assume that the exit branch in Q is strongly biased and the path C, Q, X has a frequency of 100. That is, after edge C is executed, the execution will always follow to X. We can peel off this *No*-available path, as shown in (b), effectively moving the speculation point C off this path. After peeling, the frequency of C becomes 0 and the speculation is profitable, $ex(Y) = 100 > ex(B) + ex(C) = 10 + 0$.

4 Experiments

We performed the experiments using the HP Labs VLIW back-end compiler *elcor*, which was fed `spec95` benchmarks that were previously compiled, edge-profiled, and inlined (only `spec95int`) by the *Impact* compiler. Table 1 shows program sizes in the total number of nodes and expressions. Each node corresponds to one intermediate statement. Memory requirements are indicated by the column max space, which gives the largest nodes-expressions product among all procedures. The running time of our rather inefficient implementation behaved quadratically in the number of procedure nodes; for a procedure with 1,000 nodes, the PRE time was about 5 seconds on PA-8000. Typically, the complete PRE ran faster than the subsequent dead code elimination.

Experiment 1: Disabling effects of CMP regions. The column labeled optimizable gives the percentage of expressions that reuse value along some path; 13.9% of (static) expressions have partially redundant computations. The next column prevented-CMP reports the percentage of optimizable expressions whose complete optimization by code motion is prevented by a CMP region. Code-motion PRE will fail to fully optimize 30.5% of optimizable expressions. For comparison, column prevented-POE reports expressions that will require restructuring in PoePRE.

Experiment 2: Loop invariant expressions. Next, we determined what percentage of loop invariant (LI) expressions can be removed from their invariant loops with code motion. The column loop invar shows the percentage of optimizable expressions that pass our test of loop-invariance. The following column gives the percentage of LI expressions that have a CMP region; an average of 72.5% of LI computations cannot be hoisted from all enclosing invariant loops without restructuring.

Experiment 3: Eliminated computations. The column global CSE reports the dynamic amount of computations removed by global common subexpression elimination; this corresponds to all full redundancies. The column complete PRE gives the dynamic amount of all partially redundant statements. The fact that strictly partial redundancies contribute only 1.7% (the difference between complete PRE and global CSE) may be due to the style of *Impact*'s intermediate code (e.g., multiple virtual registers for the same variable). We expect a more powerful redundancy analysis to perform better. Figure 8 plots the dynamic amount of strictly partial redundancies removed by various PRE techniques. Code-motion PRE yields only about half the benefit of a complete PRE. Furthermore, speculation results in near-complete PRE for most benchmarks, even without special hardware support (i.e., safe speculation). Speculation was carried out on the CMP as whole. Note that the graph accounts for the dynamic impairment caused by speculation.

benchmark	program size				E-1: CM prevented			E-2: loop inv		E-3: dynamic		E-4: code growth	
spec95int spec95fp	procedures	nodes (k)	expressions (k)	max space (M)	optimizable (% of *expr*)	prevented-CMP (% of *optim*)	prevented-POE (% of *optim*)	loop invar (% of *optim*)	invar-prevent (% of *LI*)	global CSE (% of all)	complete PRE (% of all)	ComPRE (% increase)	PoePRE (% increase)
099.go	372	153.6	37.3	5.8	10.2	29.6	45.4	7.1	83.4	9.5	11.7	49.9	90.2
124.m88ksim	252	79.5	17.4	4.2	13.1	32.7	45.4	13.0	78.0	7.6	9.4	30.6	59.9
126.gcc	1661	917.2	158.2	38.0	8.0	34.2	45.0	2.5	69.8	3.7	4.6	33.9	36.7
129.compress	24	3.0	0.8	0.1	13.7	20.4	43.4	9.7	45.5	11.5	14.5	14.9	20.0
130.li	357	37.4	8.4	2.0	11.8	22.4	34.4	10.4	69.9	6.8	8.0	21.5	35.1
132.ijpeg	472	81.8	22.8	1.2	13.9	24.1	45.3	5.1	78.1	4.3	5.1	48.8	104.7
134.perl	276	135.0	25.5	40.4	9.6	39.5	51.8	11.9	93.5	4.8	6.8	31.2	50.0
147.vortex	923	325.9	65.7	5.8	16.6	29.5	36.1	6.3	81.6	11.1	13.0	35.7	55.4
Avg: spec95int	542.1	216.7	42.0	12.2	12.1	29.1	43.4	8.2	75.0	7.4	9.1	33.3	56.5
101.tomcatv	1	0.8	0.2	0.2	21.4	26.4	50.9	13.2	71.4	10.2	13.3	414.1	1302.6
102.swim	7	1.6	0.6	0.1	17.0	29.2	46.2	10.4	100.0	10.7	12.0	26.7	166.0
103.su2cor	37	10.6	3.9	2.5	15.3	29.8	53.8	14.5	43.7	12.8	13.0	42.1	142.0
104.hydro2d	43	8.5	2.4	0.4	16.8	21.7	42.7	5.9	41.7	1.9	6.0	43.9	141.7
145.fpppp	37	13.6	6.7	19.6	14.6	52.2	57.7	43.0	91.9	7.1	7.7	2.4	18.2
146.wave5	110	33.3	12.3	5.3	12.4	34.8	47.8	4.9	66.2	7.1	7.8	36.6	107.6
Avg: spec95fp	39.2	11.4	4.4	4.7	16.2	32.4	49.8	15.3	69.2	8.3	10.0	94.3	313.0
Avg: spec95	326.6	128.7	25.9	9.0	13.9	30.5	46.1	11.3	72.5	7.8	9.5	59.5	166.4

Table 1: Experience with PRE based on control flow restructuring.

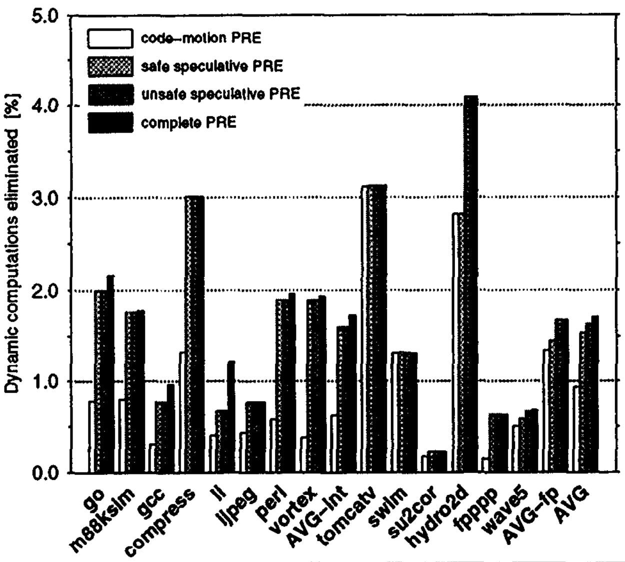


Figure 8: Benefit of various PRE algorithms: dynamic opcount decrease due to strictly partial redundancies. Each algorithm also completely removes full redundancies.

The measurements indicate that an ideal PRE algorithm should integrate both speculation and restructuring. Using restructuring when speculation would waste a large portion of benefit will provide an almost complete PRE with small code growth.

Experiment 4: Code growth. Finally, we compare the code growth incurred by ComPRE and PoePRE. To make the experiment feasible, we limited procedure size by 3,000 nodes and made the comparison only on procedures that did not exceed the limit in either algorithm. Overall, ComPRE created about three times less code growth than PoePRE.

5 Demand-Driven Frequency Analysis

Not amenable to bit-vector representation, frequency analysis [27] is an expensive component of profile-guided optimizers. We have shown that ComPRE allows restricting the scope of frequency analysis within the CMP region without a loss of accuracy. However, in large CMP regions the cost may remain high, and path profiles cannot be used as an efficient substitute when numerous hot paths fall into the region. One method to reduce the cost of frequency analysis is computing on demand only the subset of data flow solution that is needed by the optimization.

In this section, we develop a demand-driven frequency analyzer which reduces data-flow analysis time by a) examining only nodes that contribute to the solution and, optionally, b) terminating the analysis prematurely, when the solution is determined with desired precision. Besides PRE, the analyzer is suitable for optimizations where acceptable running time must be maintained by restricting analysis scope, as in run-time optimizations [5] or interprocedural branch removal [10].

Frequency analysis computes the *probability* that a dataflow fact will occur during execution. Therefore, the probability "lattice" is an infinite chain of real numbers. Because existing demand-driven analysis frameworks are built on *iterative* approaches, they only permit lattices of finite size [18] or finite height [22, 29] and hence cannot derive a frequency analyzer. We overcome this limitation by designing the demand-driven analyzer based upon *elimination* dataflow methods [28] whose time complexity is independent of the lattice shape. We have developed a demand-driven analysis framework motivated by the Allen-Cocke *interval* elimination solver [2]. Next, using the framework, a demand-driven algorithm for general frequency data-flow analysis was derived [8]. We present here the frequency solver specialized for the problem of availability.

Definitions. Assume a forward data-flow problem specified with an equation system

$$\begin{aligned} \mathbf{x}_n &= f_n(\sqcap_{m \in pred(n)} \mathbf{x}_m) \\ (x_n^1, \ldots, x_n^S) &= (f_n^1(\sqcap_{m \in pred(n)} \mathbf{x}_m), \ldots, f_n^S(\sqcap_m \mathbf{x}_m)) \end{aligned}$$

Vector $\mathbf{x}_n = (x_n^1, \ldots, x_n^S)$ is the solution for a node n, variable x_n^e denotes the fact associated with expression e. The equation system induces a dependence graph EG whose nodes are variables x_n^e and edges represent flow functions f_n^e: an edge (x_m^d, x_n^e) exists if the value of x_n^e is computed from $x_m^d, m \in pred(n)$. The graph EG is called an *exploded graph* [22]. The data flow problems underlying ComPRE are *separable*, hence x_n^e only depends on x_m^e. In value-based PRE [9], constant propagation [29], and branch correlation analysis [10], edges (x_m^d, x_n^e), $d \neq e$, may exist, complicating the analysis. The analyzer presented here handles such general exploded graphs.

Requirements. The demand-driven analyzer grew out of four specific design requirements:

1. *Demand-driven.* Rather than computing $\mathbf{x}_n$ for each node n, we determine only the desired x_n^e, i.e. the solution for expression e at a node n. Analysis speedup is obtained by further requiring that only nodes transitively contributing to the value of x_n^e are visited and examined. To guarantee worst-case behavior, when solutions for all EG nodes are desired, the solver's time complexity does not exceed that of the exhaustive Allen-Cocke method, $O(N^2)$, where N is the number of EG nodes.

2. *Lattice-independent.* The amount of work per node does not depend on lattice size, only on the EG shape.

3. *On-line.* The analysis is possible even when EG is not completely known prior to the analysis. To save time and memory, our algorithm constructs EG as analysis progresses. The central idea of on-demand construction is to determine a flow function f_n^e only when its target variable x_n^e is known to contribute to the desired solution. Furthermore, the solver must produce the solution even when EG is irreducible, which can happen even when the underlying CFG is reducible.

4. *Informed.* In the course of frequency analysis, the contribution weight of each examined node to the desired solution must be known. This information is used to develop a version of the analyzer that approximates frequency by disregarding low-contribution nodes with the goal of further restricting analysis scope.

The *exhaustive* interval data-flow analysis [2] computes $\mathbf{x}_n$ for all n as follows. First, loop headers are identified to partition the graph into hierarchic acyclic subregions, called intervals. Second, forward substitution of equations is performed within each interval to express each node solution in terms of its loop header. The substitution proceeds in the *interval order* (reverse postorder), so that each node is visited only once. Third, mutual equation dependences across loop back-edges are reduced with a *loop breaking rule* L: $\mathbf{x}_n = g(\mathbf{x}_n, \mathbf{x}_k) \rightarrow \mathbf{x}_n = L(g(\mathbf{x}_k))$. The second and third step remove cyclic dependences from all innermost loops in EG; they are repeated until all nesting levels are processed and all solutions are expressed in terms of the start node, which is then propagated to all previously reduced equations in the final propagation phase [2].

The *demand-driven* interval analysis substitutes only those equations and reduces only those intervals on which the desired x_n^e is transitively dependent. To find the relevant equations, we back-substitute equations (flow functions) into the right-hand side of x_n^e along the EG edges. The edges are added to the exploded graph *on-line*, whenever a new EG node is visited, by first computing the flow function of the node and then inserting its predecessors into the graph.

As in [2], we define an EG interval to be a set of nodes dominated by the sink of any back-edge. In an irreducible EG, a back-edge is each loop edge sinking onto a loop entry node. Because the EG shape is not known prior to analysis, on-line identification of EG intervals relies only on the structure of the underlying control flow graph. When the CFG node of an EG node x is a CFG loop entry, then x may be an EG loop entry, and we conservatively assume it is an interval head. Within each interval, back-substitutions are performed in *reversed* interval order. Such order provides *lattice-independence*, as each equation needs to be substituted only once per interval reduction, and there are at most N reductions. To find interval order on an incomplete EG, we observe that within each EG interval, the order is consistent with the reverse postorder CFG node numbering.

To loop-break cyclic dependencies along an interval back-edge, the loop is reduced before we continue into the preceding interval, recursively invoking reductions of nested loops. This process achieves demand analysis of relevant intervals. The desired solution is obtained when x_n^e is expressed exclusively using constant terms. At this point, we have also identified an EG subgraph that contributes to x_n^e, and removed from it all cyclic dependences. A forward substitution on the subgraph will yield solutions for all subgraph nodes which can be cached in case they are later desired (*worst-case running time*). This step corresponds to the propagation phase in [2], and to caching in [18, 29].

The framework instance calculates the probability of expression e being available at the exit of node n during the execution: $x_n^e = p(AVAIL_{out}[n,e] = Must) \in \mathcal{R}$. Let $p(a)$ denote the probability of edge a being taken, given its *sink* node is executed. We relate the edge probability to the sink (rather than the source, as in exhaustive analysis [27]) because the demand solver proceeds in the backward direction. The frequency flow function returns probability 1 when the node computes the expression e and 0 when it kills the expression. Otherwise, the sum of probabilities on predecessors weighted by edge execution probabilities is returned. Predicates *Comp* and *Transp* are defined in Figure 3.

$$x_n^e = \begin{cases} 1.0 & \textbf{if } Comp(n,e) \wedge Transp(n,e), \\ 0.0 & \textbf{if } \neg Transp(n,e), \\ \displaystyle\sum_{m \in pred(n)} p((m,n)).x_m^e & \textbf{otherwise.} \end{cases}$$

The demand frequency analyzer is shown in Figure 9. Two data structures are used : *sol* accumulates the constant terms of the desired probability x_n^e; *rhs* is the current right-hand side of x_n^e after all back-substitutions. The variables *sol* and *rhs* are organized as a stack, the top being used in the currently analyzed interval. The algorithm treats *rhs* both as a symbolic expression and as a working set of pending nodes (or yet unsubstituted variables, to be precise). For example, the value of *rhs* may be $0.25*m+0.4*k$, where the weights are contributions of nodes m and k to the desired probability x_n^e. If e is never available at m, and is available at k with probability 0.5, then it is available at node n with probability $0.25 * 0 + 0.4 * 0.5 = 0.2$. More formally, the contribution weight of a node represents the probability that a path from that node to n without a computation or a kill of the expression e will be executed.

First, the *rhs* is set to $1.0 * n$ in line 1. Then, flow functions are back-substituted into *rhs* in post-order (line 3). Substitutions are repeated until all variables have been replaced with constants (line 2), which are accumulated in *sol*. If a substituted node x computes the expression e, its weight $rhs[x]$ is added to the solution and x is removed from the *rhs* by the assignment $rhs[x] := 0.0$ (line 6). In the simple case when x is not a loop entry node (line 12), its contribution c is added to each predecessor's contribution, weighted by the edge probability p. If x is a loop entry node (line 8), then before continuing to the loop predecessor, all self-dependences of x are found in a call to **reduce_loop**. The procedure **reduce_loop** mimics the main loop (lines 1–5) but it pushes new entries on the stacks to initiate a reduction of a new interval and also marks the loop entry node to stop when back-substitution collected cyclic dependences along all cyclic paths on the back-edge edge (y, x). The result of **reduce_loop** is returned in a *sol-rhs* pair (s, r), where s is the constant and r the set of unresolved variables, e.g. $x = r + s = 0.3x + 0.1$. If EG is reducible, the set r contains only x. The value $r[x] = 0.3$ is the weight of the x's self-dependence, which is removed by the loop breaking rule derived for frequency analysis from the sum of infinite geometric sequence (lines 10–11). After the algorithm terminates, the stack *visited* (line 14) specifies the order in which forward substitution is performed to cache the results. Also shown in Figure 9 is an execution trace of the demand-driven analysis. It computes the probability that the expression computed in nodes F, H, and killed in A, D, is available at node C. All paths where availability holds are highlighted.

Approximate Data-Flow Analysis. Often, it is necessary to sacrifice precision of the analysis for its speed. We define here a notion of *approximate* data flow information, which allows the analyzer a predetermined degree of conservative imprecision. For example, given a 5% imprecision level ($\epsilon = 0.05$), the analyzer may output "available: 0.7," when the maximal fixed point solution is "available: 0.75." The intention of permitting underestimation is to reduce the analysis cost. When the analyzer is certain that the contribution of a node (and all its incoming paths) to the overall solution is less than the imprecision level, it can avoid analyzing the paths and assume at the node the most conservative data-flow fact.

Because the algorithm in Figure 9 was designed to be *informed*, it naturally extends to approximate analysis. By knowing the precise contribution weight of each node as the analysis progresses, whenever the sum of weights in *rhs* at the highest interval level falls below ϵ (the while-condition in line 2), we can terminate and guarantee the desired precision. An alternative scenario is more attractive, however. When a low-weight node is selected in line 3, we throw it away. We can keep disregarding such nodes until their total weights exceed ϵ. In essence, this approach performs analysis along hot paths [4], and on-line region formation [21].

The idea of terminating the analysis before it could find the precise solution was first applied in the implementation of interprocedural branch elimination [10]. Stopping after visiting a thousand nodes resulted in two magnitudes of analysis speedup, while most optimization opportunities were still discovered. However, without the *approximate* frequency analyzer developed in this paper, we were unable to a) determine the benefit of restructuring, b) select a profitable subset of nodes to duplicate, and c) get a bound on the amount of opportunities lost due to early termination.

Algorithm complexity. In an arbitrary exploded graph, **reduce_loop** may be (recursively) invoked on each node. Hence, each node may be visited at most N_E times, where $N_E = NS$ is the number of EG nodes, N the number of CFG nodes, and S the number of optimized expressions. With caching of results, then each node is processed in at most one invocation of the algorithm in Figure 9, yielding worst-case time complexity of $O(N_E^2) = O(N^2S^2)$. Since real programs have loop nesting level bound by a small constant, the expected complexity is (NS), as in [2].

Although most existing demand-driven data-flow algorithms ([18, 22], [29] in particular) can be viewed (like ours) to operate on the principle of back-substituting flow functions into the right-hand side of the target variable, they do not focus on specifying a profitable order of substitutions (unlike ours) but rely instead on finding the fixed point iteratively. Such an approach fails on infinite-height lattices where CFG loops keep always iterating towards a better approximation of the solution. Note that breaking each control flow cycle by inserting a widening operator [13] does not appear to resolve the problem because widening is a local adjustment primarily intended to approximate the solution. Therefore, in frequency analysis, too many iterations would be required to achieve an acceptable approximation. Instead of fixing the equation system locally, a global approach of structurally identifying intervals and reducing their cyclic dependences seems necessary. We have shown how to identify intervals and perform substitutions in interval order on demand, even when the exploded graph is not known prior to the analysis. We believe that existing demand methods can be extended to operate in a structural manner, enabling the application of loop-breaking rules. This would make the methods reminiscent of the elimination algorithms [28].

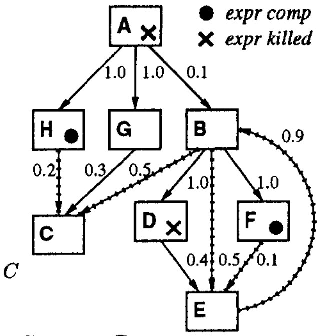
6 Conclusion and Related Work

The focus of this paper is to improve program *transformations* that constitute value-reuse optimizations commonly known as Partial Redundancy Elimination (PRE). In the long history of PRE research and implementation, three distinct transformations can be identified. The seminal paper by Morel and Renviose [26] and its derivations [11, 14, 15, 16, 17, 24] employ pure, non-speculative *code motion*. Second, the complete PRE by Steffen [30] is based on control flow *restructuring*. Third, navigated by path profile information, Gupta et al apply *speculative code motion* in order to avoid code-motion obstacles by controlled impairment of some paths [20].

In this work, we defined the *code-motion-preventing* (CMP) region, which is a CFG subgraph localizing adverse effects of control flow on the desired value reuse. The notion of the CMP is applied to enhance and integrate the three existing PRE transformations in the following ways, **1.** Code motion and restructuring are integrated to remove all redundancies at minimal code growth cost (ComPRE). **2.** Morel and Renviose's original method is expressed as a restricted (motion-only) case of the complete algorithm (CM-PRE). **3.** We develop an algorithm whose power adjusts contin-

```
Input:     node n, expression e.
Output:    in sol, the probability of e being available at the exit of n.
sol :      stack of reals (names sol, rhs refer always to top of stack)
rhs :      stack of sets of unsubstituted nodes n with weights rhs[n]
post-dfs : post-order numbering of CFG nodes

1   sol := 0; rhs := {}; rhs[n] := 1.0
2   while rhs not empty do
3       select from rhs a node x with smallest post-dfs(x)
4       substitute(x)
5   end while
    procedure substitute(node x)
        if x has not been visited, determine its flow function
        if x computes or kills e, adjust sol and remove x from rhs
        if Comp(x,e) ∧ Transp(x,e) then
6           sol := sol + rhs[x]; rhs[x] := 0.0; return
7       else if ¬Transp(n,e) then rhs[x] := 0.0; return
        back-edge is each edge that meets a loop-entry edge
8       if back-edge (y,x) exists then  assume one back-edge per node
            substitute for y until x occurs on the r.h.s.
9           (s,r) := reduce_loop(y,x)
            apply loop breaking rule: sum of infinite geom. sequence
10          c := rhs[x]/(1 − r[x])
11          rhs := rhs + c × r; sol := sol + c × s
12      else c := rhs[x]
        substitute "acyclic" predecessors
        for each non-backedge node z ∈ pred(x) do
13          rhs[z] := rhs[z] + c × p((z,x))
        end for
        x is now fully substituted
14      rhs[x] := 0.0; visited.push(x)
    end substitute
    function reduce_loop(node u, node v)
15      mark v; sol.push(0); rhs.push({}); rhs[u] := p((u,v))
16      while rhs contains unmarked nodes do
17          select from rhs an unmarked node x with lowest post-dfs(x)
18          substitute(x)
19      end while
20      unmark v; return (sol.pop(), rhs.pop())
    end reduce_loop
```




```
post-dfs: H, G, F, ...
Input: n = C
Output: p = 0.2818
1   sol := 0; rhs = 1.0 * C
4   substitute(C)
12  c := 1.0
13  rhs := 0.2 * H + 0.3 * G + 0.5 * B
4   substitute(H)
6   sol := sol + rhs[H] := 0 + 0.2
6   rhs = 0.3 * G + 0.5 * B
4   substitute(G)
12  c := 0.3
13  rhs := 0.5 * B + 0.3 * A
4   substitute(B)
9       reduce_loop(E, B)
15      mark B; sol_1 := 0; rhs_1 := 0.9 * E
18      substitute(E)
12      c := 0.9;
13      rhs_1 := 0.36 * D + 0.45 * B + 0.09 * F
18      substitute(F)
6       sol_1 := sol_1 + rhs_1[F] := 0 + 0.09
6       rhs_1 = 0.36 * D + 0.45 * B
18      substitute(D)
7       rhs_1 := 0.45 * B
20      unmark B; return (0.09, 0.45 * B)
10  c := 0.5/(1 − 0.45) = 0.91
11  rhs := 0.5 * B + 0.3 * A + 0.91 * 0.45 * B
11  sol := 0.2 + 0.91 * 0.09 := 0.2818
13  rhs := 0.91 * B + 0.3 * A + 0.91 * 0.1 * A
14  rhs := 0.391 * A
4   substitute(A)
7   sol := 0.2818 unchanged ← Final probability
7   rhs := 0.0
```

Figure 9: Demand-driven frequency analysis for availability of computations, and a trace of its execution.

ually between the motion-only and the complete PRE in response to the program profile (PgPRE). **4.** We demonstrate that speculation can be navigated precisely by edge profiles alone. **5.** Path profiles are used to integrate the three transformations and balance their power at the level of CMP paths.

While PRE is significantly improved through effective program transformations presented in this paper, a large orthogonal potential lies in *detecting* more redundancies. Some techniques have used powerful analysis to uncover more value reuse than the traditional PRE analysis [9, 11]. However, using only code motion, they fail to completely exploit the additional reuse opportunities. Thus, the transformations presented here are applicable in other styles of PRE as well, for example in elimination of loads.

Ammons and Larus [4] developed a constant propagation optimization based on restructuring, namely on peeling of hot paths. In their analysis/transformation framework, restructuring is used not only to exploit optimization opportunities previously detected by the analysis, as is our case, but also to improve the analysis precision by eliminating control flow merges from the hot paths. Even though our PRE cannot benefit from hot path separation (our distributive data-flow analysis preserves reuse opportunities across merges), a more complicated analysis (e.g., redundancy of array bound checks) would be improved by their approach. After the analysis, their algorithm recombines separated paths that present no useful opportunities. It is likely that path recombination can be integrated with code motion, as presented in this paper, to further reduce the code growth.

In a global view, we have identified four main issues with *path-sensitive* program optimizations [8]: a) solving *non-distributive* problems without conservative approximation (e.g. non-linear constant propagation), b) collecting *distinct opportunities* (e.g., variable has different constant along each path), c) *exploiting* distinct opportunities (e.g., enabling folding of path-dependent constants through restructuring), and d) directing the analysis effort towards hot paths. In the approach of Ammons and Larus, all four issues are attacked uniformly by separation of hot paths, their subsequent individual analysis, and recombination. Our approach is to reserve restructuring for the actual transformation. This implies a different overall strategy: a) we solve non-distributive problems precisely along *all* paths by customizing the data-flow *name space* [9], b) we collect distinct opportunities through demand-driven analysis as in branch elimination [10], which is itself a form of constant propagation, c) we exploit all profitable opportunities with economical transformations, and d) avoid infrequent program regions using the approximation frequency analysis (the last three presented in this paper).

7 Acknowledgments

We are indebted to the *elcor* and *Impact* compiler teams for providing their experimental infrastructure. Sadun Anik, Ben-Chung Cheng, Brian Dietrich, John Gyllenhaal, and Scott Mahlke provided invaluable help during the implementation and experiments. Comments from Glenn Ammons, Evelyn Duesterwald, Jim Larus, Mooly Sagiv, Bernhard Steffen, and the anonymous reviewers helped to improve the presentation of the paper. This research was partially supported by NSF PYI Award CCR-9157371, NSF grant CCR-9402226, and a grant from Hewlett-Packard to the University of Pittsburgh.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison Wesley, 1986.

[2] F. E. Allen and J. Cocke. A program data flow analysis procedure. *CACM*, 19(3):137–147, March 1976.

[3] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM SIGPLAN '97 Conf. on Prog. Language Design and Impl.*, pages 85–96, 1997.

[4] Glenn Ammons and James L. Larus. Improving data-flow analysis with path profiles. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, 1998.

[5] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 149–159, 21– May 1996.

[6] A. Ayers, R. Schooler, and R. Gottlieb. Aggressive inlining. In *Proceedings of the ACM SIGPLAN '97 Conf. on Prog. Language Design and Impl.*, pages 134–145, June 1997.

[7] Thomas Ball and James R. Larus. Efficient path profiling. In *29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 46–57, 1996.

[8] Rastislav Bodik. *Path-Sensitive Compilation.* PhD thesis, University of Pittsburgh, in preparation.

[9] Rastislav Bodik and Sadun Anik. Path-sensitive value-flow analysis. In *Conference Record of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1998.

[10] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Interprocedural conditional branch elimination. In *Proceedings of the ACM SIGPLAN '97 Conf. on Prog. Language Design and Impl.*, pages 146–158, June 1997.

[11] Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 159–170, June 1994.

[12] F. Chow, S. Chan, R. Kennedy, S.-M. Liu, R. Lo, and P. Tu. A new algorithm for partial redundancy elimination based on SSA form. In *Proceedings of the ACM SIGPLAN '97 Conf. on Prog. Language Design and Impl.*, pages 273–286, June 1997.

[13] P. Cousot and R. Cousot. Abstract intrepretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

[14] Dhanajay M. Dhamdhere, Barry K. Rosen, and Kenneth F. Zadeck. How to analyze large programs efficiently and informatively. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 212–223, July 1992.

[15] Dhananjay M. Dhamdhere. Practical adaptation of the global optimization algorithm of Morel and Renvoise. *ACM Transactions on Programming Languages and Systems*, 13(2):291–294, April 1991.

[16] K. Drechsler and M. 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, October 1988.

[17] K. Drechsler and M. Stadel. A variation of Knoop, Rüthing, and Steffen's *lazy code motion*. *ACM SIGPLAN Notices*, 28(5):635–640, May 1993.

[18] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems*, 19(6):992–1030, November 1997.

[19] R. Gupta, D. Berson, and J.Z. Fang. Resource-sensitive profile-directed data flow analysis for code optimization. In *30th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 358–368, December 1997.

[20] R. Gupta, D. Berson, and J.Z. Fang. Path profile guided partial redundancy elimination using speculation. In *IEEE International Conference on Computer Languages*, May 1998.

[21] Richard E. Hank, Wen-Mei W. Hwu, and B. Ramakrishna Rau. Region-based compilation: An introduction and motivation. In *28th Annual IEEE/ACM International Symposium on Microarchitecture*, Ann Arbor, Michigan, 1995.

[22] Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand Interprocedural Dataflow Analysis. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 104–115, October 1995.

[23] Johan Janssen and Henk Corporaal. Controlled node splitting. In *Compiler Construction, 6th International Conference*, volume 1060 of *Springer Lecture Notes in Computer Science*, pages 44–58, Sweden, April 1996.

[24] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Optimal code motion: Theory and practice. *ACM Trans. on Progr. Languages and Systems*, 16(4):1117–1155, 1994.

[25] S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W.M. W. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel scheduling for VLIW and superscalar processors. *ACM Transactions on Computer Systems*, 11(4):376–408, 1993.

[26] E. Morel and C. Renviose. Global optimization by supression of partial redundancies. *CACM*, 22(2):96–103, 1979.

[27] G. Ramalingam. Data flow frequency analysis. In *Proceedings of the ACM SIGPLAN '96 Conf. on Progr. Language Design and Implementation*, pages 267–277, June 1996.

[28] Barbara G. Ryder and Marvin C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):277–316, September 1986.

[29] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1–2):131–170, 1996.

[30] Bernhard Steffen. Property oriented expansion. In *Proc. Int. Static Analysis Symposium (SAS'96)*, volume 1145 of *LNCS*, pages 22–41, Germany, September 1996. Springer.