E-path_PRE—Partial Redundancy Elimination Made Easy

Dhananjay M. Dhamdhere
dmd@cse.iitb.ac.in
Department of Computer Science and Engineering
Indian Institute of Technology, Mumbai 400 076 (India).

Abstract

Partial redundancy elimination (PRE) subsumes the classical optimizations of loop invariant movement and common subexpression elimination. The original formulation of PRE involved complex bi-directional data flows and had two major deficiencies—missed optimization opportunities and redundant code movement. To eliminate redundant code movement, most current PRE approaches use a hoisting-followed-by-sinking approach. Unfortunately, this approach has a high conceptual complexity and requires complicated correctness proofs.

We show that optimization by partial redundancy elimination is simpler than it has been made out to be. Its essence is the concept of *eliminatability* of an expression. We show that E-path_PRE, a formulation of PRE based on the concept of eliminatability paths (E-paths), is easy to understand and simple to prove correct. It uses only well-known data flow concepts of available expressions and anticipatable (i.e. very-busy) expressions to directly identify code insertion points which avoid redundant code movement. These features reduce the conceptual complexity of PRE considerably. Interestingly, performance studies show that E-path_PRE is also less expensive to perform than the closest equivalent approach to PRE. This is a sheer bonus.

Keywords Partial redundancy elimination, eliminatability of expressions, code optimization, data flow analysis, redundant code movement.

1 Introduction

Partial redundancy elimination (PRE) is a powerful optimization technique which integrates classical optimizations of loop invariant movement and common subexpression elimination. The original formulations of PRE by Morel and Renvoise [25]—hereafter called MRA—and by Dhamdhere and Isaac [9] used a combination of movement of expression evaluations existing in a program and insertions of new evaluations of an expression to suppress partial redundancies in a program. MRA performed PRE using static analysis of programs, while the approach by Dhamdhere and Isaac depended on execution frequency information obtained by profiling a program. Much of PRE work since these original formulations has broadly followed the approach of MRA.

MRA involved complex bi-directional data flows which were hard to understand. It also suffered from two deficiencies [4]. It missed some opportunities of optimization. It also performed redundant code movement—that is, code movement which did not improve execution speed along any path in a program. This effect was manifest in two forms, either code movement occurred even when redundancies were not eliminated, or code insertions were performed farther away from a redundancy site than necessary. Redundant code movement unnecessarily extended lifetimes of compiler generated temporary variables and registers.

Many researchers have tried to improve the formulation of PRE [4, 12, 13, 14, 15, 19, 20, 21, 32] or to enhance its scope to elimination of *all* partial redundancies in a program using code expansion and restructuring [2, 31]. The basic PRE framework has also been extended to include PRE of assignments [8, 23], code size efficient PRE [30], strength reduction optimization [6, 13, 16, 18] and to use it for other applications like live range determination in register assignment [5, 7, 24].

The approaches aimed at improving the formulation of PRE have focused on three aspects

1. Elimination of deficiencies of MRA
2. Simplification of PRE data flows
3. Reduction in solution complexity of PRE data flows.

MRA missed many optimization opportunities because it confined insertion of expressions to nodes in a program flow graph. Dhamdhere [4] proposed a code placement model which permitted insertion of expressions in nodes as well as along edges in a program flow graph. Use of this model was shown to capture the optimization possibilities missed by MRA. It also permitted the bi-directional data flow problems involved in MRA to be decomposed into two unidirectional data flow problems. This decomposition was believed to simplify the data flows involved in PRE, and reduce their solution complexity.

To suppress redundant code movement, the approach in [4] advocated that PRE be viewed as a hoisting problem to eliminate partial redundancies followed by a sinking problem to eliminate redundant code movement. This *hoisting-followed-by-sinking* approach has been used in most subsequent PRE work. Its influence has been strengthened by the lazy code motion (LCM) approach formulated by Knoop, Ruthing and Steffen [21] and modified by Drechsler and Stadel [15] and Ruthing [29]. Even the work by Dhaneshwar and Dhamdhere [13], where some of the concepts used in the current work originate, implemented their approach to PRE using a hoisting-followed-by-sinking approach. The static single assignment (SSA) based implementation of PRE reported in [3] also uses this approach. Papers which do not use the hoisting-followed-by-sinking approach are [2, 27, 32]. Of these, [32] does not address redundant code movement, while [2, 27] suffer from a peculiar form of redundant code movement wherein an expression may be hoisted from a node and inserted into all its in-edges.

Benefits concerning simplification of PRE data flows and reduction in their solution complexity have been illusory. Khedker and Dhamdhere [10, 19] showed that solution of bi-directional data flows was no more complex than solution of unidirectional data flows, hence solution complexity is not reduced by using unidirectional data flows instead of bi-directional data flows. Also, conceptual complexity of the hoisting-followed-by-sinking approach is very high, which makes PRE algorithms hard to understand and implement.

1.1 A Conceptual Basis for PRE

None of the papers in the PRE literature formulate a basis for PRE through code movement and code insertion, i.e. none of them provide a simple answer to the question: Under what conditions can a partially redundant occurrence of an expression be removed by a PRE approach based on insertion and movement of expressions? The absence of a conceptual basis has made PRE very difficult to understand and practice. It has also led to some deficiencies mentioned in Section 3.4.

In reality, PRE is simpler than it has been made out to be. Its essence is the concept of *eliminatability* of an expression. We show that a formulation of PRE based on the concept of *eliminatability paths* (E-paths) leads to a PRE approach which is easy to understand and simple to prove correct.

An overview of our approach using the terminology of Section 2.1 is as follows: An occurrence of expression e in a program is *eliminatable* if an E-path ending at the expression exists in the program. Let $[b_i \ \ldots \ b_k]$ be an E-path. This path contains occurrences of e in nodes b_i and b_k which are downwards and upwards exposed [1, 26], respectively, and e is either available or anticipatable (i.e. very busy) at the exit of each node in the path $[b_i \ldots b_k)$. We eliminate the occurrence of e in node b_k using a temporary variable t_e. We save the value of e in t_e in node b_i and insert computations $t_e \leftarrow e$ in each node b_h or edge (b_h, b_j) such that b_h is a predecessor of some node in the path $(b_i \ \ldots \ b_k]$ and b_h does not itself lie along an E-path for e. We name this PRE approach as *E-path_PRE*.

The key advantage of E-path_PRE is its simplicity. It uses only well-known data flow concepts of available expressions and anticipatable (i.e. very-busy) expressions [1, 26] to identify eliminatable expressions in a program. Unlike earlier approaches [4, 11, 12, 13, 15, 21], it does not use, even conceptually, the hoisting-followed-by-sinking approach to avoid redundant code movement. These aspects considerably reduce the conceptual complexity of PRE. Another advantage is its ability to handle basic blocks containing more than one statement. This fact extends its applicability to a larger class of program graphs than the PRE approaches of [2, 21].

Although our main claim is simplicity of E-path_PRE, it is interesting to note that E-path_PRE also provides better practical performance compared to an implementation of a variation of lazy code motion reported in [15]. The key data flow which identifies an *E-path_suffix* is almost five times less expensive in terms of bit-vector operations in a worklist iterative solution technique than the *Later* data flow in [15], and about 1.75 times less expensive in terms of number of iterations of a round-robin iterative solution technique. Overall, E-path_PRE is 36.7 percent less expensive, i.e. over 1.5 times less expensive, in terms of bit-vector operations in a worklist iterative solution technique. These reductions are significant even though performance of an optimizer would depend on several other factors like overhead of managing worklists or iterations.

We describe fundamentals of the eliminatability path approach to PRE in Section 2. Section 3 presents details

of E-path_PRE and design of its data flows. Section 3.4 compares our approach with the approaches in [13, 15, 21]. The Appendices contain proofs of E-path_PRE's properties and performance statistics.

2 PRE Using Eliminatability Paths

We assume that a program is represented in the form of a program flow graph $G \equiv (N, E, n_0)$, where N is the set of basic blocks, E is the set of control flow edges and n_0 is the entry node of the program [1, 26]. An occurrence of e in a node b_k is eliminatable if an *eliminatability path* (E-path) $b_i \ldots b_k$ exists in the program. An E-path is a formalization of a similar but unnamed concept developed in Dhaneshwar and Dhamdhere [13].

2.1 Terminology

An expression e is *locally available* in node b_i if b_i contains a *downwards exposed* occurrence of e, i.e. an occurrence of e which is not followed by a definition of any of its operands. An expression e is *available* (*partially available*) at a program point if along all paths (along some path) from the entry node n_0 to that point, there exists a computation of e not followed by a definition of any of its operands. A computation of e at program point w is *redundant* if e is available at w, and *partially redundant* if it is partially available at w.

An expression e is *locally anticipatable* in node b_k if b_k contains an *upwards exposed* occurrence of e, i.e. an occurrence of e which is not preceded by a definition of any of its operands. e is *anticipatable* at a program point if each path starting at that point contains a computation of e not preceded by a definition of any of its operands.

Optimization by code movement should not change the execution behavior of a program by raising exceptions which would not have occurred in the original program. An expression e is *safe* at a point if it is either anticipatable or available at that point [17]. An optimization algorithm involving code movement should place computations of e only at points where e is safe.

A node b_k is *empty* with respect to an expression e if b_k does not contain an occurrence of e, or definition(s) of any of its operands. $empty(b_i)$ indicates that node b_i is empty with respect to e. $[b_i \ldots b_k]$ denotes a path from b_i to b_k which includes both b_i and b_k, whereas $[b_i \ldots b_k)$ denotes a path which includes b_i but does not include b_k. $(b_i \ldots b_k]$ and $(b_i \ldots b_k)$ are analogously defined. $empty([b_i \ldots b_k])$ indicates $empty(b_j)$ for all b_j in the path $[b_i \ldots b_k]$, etc.

Definition 1 (E-path) *An* E-path *for an expression e is a path $b_i \ldots b_k$ in G such that*

a. e is locally available in b_i, and locally anticipatable in b_k.

b. empty$((b_i \ldots b_k))$,

c. e is safe at the exit of each node on the path $[b_i \ldots b_k)$.

Predicate $start(b_j)$ ($end(b_j)$) indicates if b_j is a start node (end node) of an E-path. $in_e\text{-}path(b_j)$ indicates if b_j exists in an E-path $[b_i \ldots b_k]$. The first occurrence of e in the end node of an E-path is an eliminatable occurrence. As mentioned in Section 1, we save the value of e in a temporary variable t_e in node b_i and insert computations $t_e \leftarrow e$ in each node b_h or edge (b_h, b_j) such that b_h is a predecessor of some node in the path $(b_i \ \ldots \ b_k]$ and b_h does not itself lie along an E-path for e.

2.2 Optimization Using E-paths

To perform partial redundancy elimination of an expression e, the compiler introduces a variable t_e and performs the following actions:

1. *Save the value of e:* A computation $t_e \leftarrow e$ is inserted before an occurrence of e and the occurrence of e is replaced by t_e.

2. *Insert an evaluation of e:* A computation $t_e \leftarrow e$ is inserted.

3. *Eliminate a redundant evaluation of e:* An occurrence of e is replaced by t_e.

Algorithm 1 identifies points in G where e should be saved, inserted or eliminated.

Algorithm 1 (PRE using E-paths)

For all nodes b_j in G such that $in_e\text{-}path(b_j)$

1. *Identify insertion points:* If $\neg start(b_j)$, for every edge (b_h, b_j) such that $\neg in_e\text{-}path(b_h)$
   (a) insert $t_e \leftarrow e$ in node b_h if $in_e\text{-}path(b_s)$ for all successors b_s of b_h
   (b) insert $t_e \leftarrow e$ along edge (b_h, b_j) if the condition in Step 1a is not satisfied.
2. *Identify save points:* If $start(b_j)$ and $\neg(end(b_j) \wedge b_j$ does not contain a definition of an operand of e), save the value of last occurrence of e in node b_j in variable t_e.
3. *Identify redundant occurrences of an expression:* If $end(b_j)$, the first occurrence of e in b_j is replaced by t_e.

A save point is located in a start node. However, if $start(b_j)$ and $end(b_j) \wedge b_j$ does not contain definitions of e's operands, t_e already contains the value of e. Hence Step 2 does not save the value of e in t_e in such nodes.

Code insertion performed in Step 1b is called *edge placement.* Edge placement was mentioned in [25] but was not used in MRA. It was independently integrated into PRE algorithms by [4], [14], and [28]. Advantage of inserting a computation in edge (b_h, b_j) is that it is removed from the E-path without being inserted into paths which pass through b_h but do not pass through b_j. To perform edge placement along edge (b_h, b_j), a synthetic node b_{hj} is placed along the edge, and $t_e \leftarrow e$ is placed in this node [4, 13]. (Placement of a synthetic node in an edge is called *edge-splitting.*)

Correctness of the optimization performed by Algorithm 1 can be informally argued as follows: Consider an E-path $b_i \dots b_k$. e is available at the exit of b_i. For each edge (b_h, b_j) such that b_j lies along the E-path, e is available along edge (b_h, b_j) in the optimized program due to Step 1. Hence e is available at entry to b_k in the optimized program.

Absence of redundant code movement is easy to establish. Let e be inserted in a node b_l (original or synthetic node) which is a predecessor of node b_j lying along an E-path $b_i \dots b_k$. Insertion in b_l involves redundant code movement if e could have been inserted in b_j without loss of computational efficiency. However, if e were to be placed in b_j, $b_i \dots b_j$ would be an E-path according to Def. 1, and e would be eliminatable in b_j. Thus Algorithm 1 does not perform redundant code movement.

2.3 An Example

Figure 1 contains an example of optimization using Algorithm 1. Three E-paths exist in the program for $a*b$: b_8-b_9, b_9-b_8 and b_9-b_{10}-b_{11}. Path b_2-b_4-b_8 is not an E-path since $a*b$ is neither available nor anticipatable in node b_4. E-path b_8-b_9 does not have any entry edges, hence no insertion is necessary to eliminate the occurrence of $a*b$ in node b_9. While optimizing the E-path b_9-b_8, $t_1 \leftarrow a*b$ is inserted along edge (b_4, b_8). E-path b_9-b_{10}-b_{11} is optimized by inserting $t_1 \leftarrow a*b$ in node b_7. Evaluations of $a*b$ in nodes b_8, b_9 and b_{11} are replaced by t_1. Note that nodes b_8 and b_9 are both start and end nodes. Hence value of $a*b$ is not saved in these nodes. Two E-paths b_2-b_4-b_8-b_9-b_{10} and b_3-b_4-b_8-b_9-b_{10} exist for $c*d$. No insertions are necessary. Value of $c*d$ is saved in t_2 in b_2, b_3, and the evaluation of $c*d$ in node b_{10} is replaced by t_2.

MRA [25] would not eliminate $a*b$ from node b_8 because it performs code insertions only in nodes and insertion in node b_4 is not safe. Thus, it would fail to optimize $a*b$ of node b_8. MRA would delete $a*b$ from node b_9. It would also insert $a*b$ in node b_5 and delete it from node b_{11}. This is an instance of redundant code movement because insertion in node b_5 instead of node b_7 does not provide any execution benefits. MRA would optimize $c*d$ as shown in Fig. 1 (except that it does not contain any provision to identify save points of nodes b_2, b_3—they would have to be identified by an additional data flow problem).

3 Identification of E-paths

We analyse features of an E-path and design simple unidirectional data flow problems to identify E-paths. Figure 3 defines the data flow properties used.

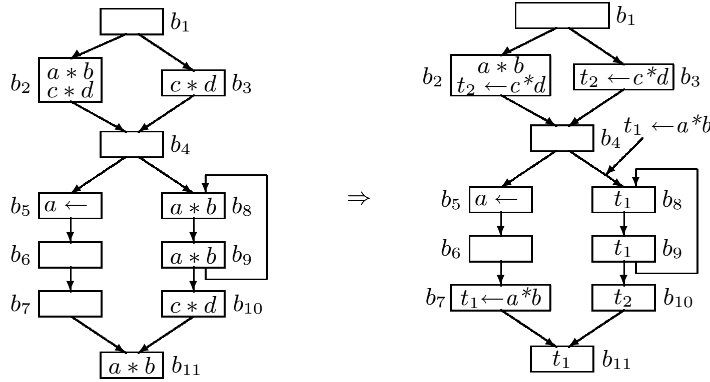

Figure 1: Optimization using e-paths

3.1 Regions of Availability & Anticipatability

From Part (c) of Def. 1, e is either available or anticipatable at the exit of every node b_j in $[b_i \dots b_k)$. Hence the E-path $b_i \dots b_k$ can be divided into two parts—an *availability part* such that e is available at the exit of each node in the part, and an *anticipatability part* such that e is anticipatable at the entry of each node in the part. Note that these two parts can overlap, i.e. e may be both available at the exit and anticipatable at the entry of some nodes.

Let R_{av} and R_{ant} be regions in G containing nodes with e available at exit and anticipatable at entry, respectively. Figure 2 shows five paths in G. Each path n is of the form $b_i^n \dots b_k^n$. Nodes b_k^2, b_k^3 and b_k^5 contain a locally anticipatable occurrence of e, and nodes b_i^1, b_i^2 and b_i^3 contain a locally available occurrence of e. Paths 2, 3 and 4 contain both availability and anticipatability parts. $R_{av} \bigcap R_{ant} = \phi$ if such paths do not exist.

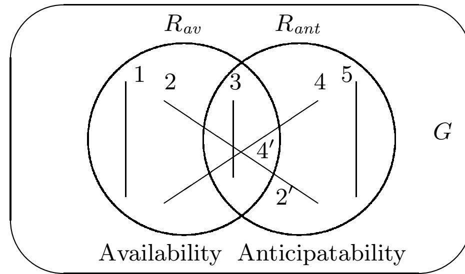

Figure 2: Availability and anticipatability regions

Path 1 is not an E-path. Path 4 cannot be an E-path since e is not locally available at the exit of b_i^4, however its segment $b_i^{4'} \dots b_k^{4'}$ such that $b_i^{4'}$ is its first node in R_{av} and $b_k^{4'}$ is the last node in R_{ant} is an E-path since e must be locally available in $b_i^{4'}$ and locally anticipated in $b_k^{4'}$. This path is designated as path $4'$. It is analogous to path 3. Path 5 cannot be an E-path. If b_i^5 contained a locally available occurrence of e, path 5 would be analogous to path 2.

Consider a node b_j in the availability part of an E-path $b_i \dots b_k$ such that $b_j \neq b_i$. Every predecessor b_h of b_j must be in R_{av} (see path 2, 3 or $4'$ in Fig. 2). Hence b_h-$b_j \dots b_k$ exists in some E-path $[b_l \dots b_k]$, and b_h does not satisfy the conditions for insertion of e in Step 1 of Algorithm 1. Hence insertion of e may be performed only in predecessors of nodes which are included in the anticipatability part but not included in the availability part of an E-path. We introduce the concept of an *E-path suffix* to designate part of an E-path which contains such nodes.

Definition 2 (E-path suffix) *An* E-path suffix *is the maximal suffix of an E-path such that $Ant_in \cdot \neg Av_in =$ true for each node in it.*

From Def. 1 it is seen that the first node of an E-path suffix has a predecessor with $Av_out =$ *true*, and the last node is the end-node of the E-path, i.e. a node with $Antloc =$ *true*. In Fig. 2, part $2'$ is the suffix of E-path 2.

Note that the suffix of an E-path may be null. In paths 3 and $4'$, the E-path suffix is null. Hence the optimization algorithm will not find nodes b_h, b_j such that $t_e \leftarrow e$ needs to be inserted in b_h or along (b_h, b_j).

3.2 Data Flows for E-path_PRE

Figure 3 shows the data flows used in E-path_PRE. Here we use the convention that the data flow value 1 implies that a data flow property is *true* and 0 implies that it is *false*. Since the data flow value for each expression can be represented in one bit, we use a bit vector to represent data flow values of all expressions at entry/exit of a node in G. For a Π problem 1 implies TOP and 0 implies BOT, where TOP and BOT are the lattice $\top$ and $\perp$ elements for a single bit lattice. For a Σ problem 1 implies BOT and 0 implies TOP. A data flow problem is solved using conventional initializations, i.e. entry of n_0 (exit of an exit node) is initialized to $0\ldots0$, which is a bit vector containing all 0s, for a forward (backward) data flow problem and all other values are initialized to $\top$ [1, 19]. This initialization ensures a maximal fixed point of the data flows. If a program contains n nodes and r is the bit vector size, the complexity of data flow analysis is $O(n \cdot r)$ for a worklist iterative technique and $O(n^2 \cdot r)$ for a round-robin iterative technique.

Data flow equations (1)–(4) in Fig. 3 compute *Av_in/out* and *Ant_in/out* of expressions in the nodes of G. We use the *Av_out* and *Ant_in* properties to identify an E-path suffix. As observed in Section 3.1, the first node of an E-path suffix has a predecessor with *Av_out* = *true*. Hence *Eps_in* is set to *true* for a node in which e is anticipatable but not available and the node is a successor of a node which has e available at its exit. The last node of an E-path suffix is the end-node of an E-path. Hence $Eps_out_i = Eps_in_i \cdot \neg Antloc_i$.

If an E-path contains an E-path suffix, $Eps_in_i \cdot Antloc_i = true$ identifies its end node, else $Av_in_i \cdot Antloc_i = true$ identifies it. Hence $Redund_i$ is set to *true* in these two cases. Insertions are performed as follows: If node b_j with $Eps_in_j = true$ has a predecessor b_i with $Eps_out_i = false$ and $Av_out_i = false$, $Insert_i = true$ if $Eps_in = true$ for all successors of b_i, else $Insert_{ij} = true$. This action implements Step 1 of Algorithm 1.

To compute the property $Save_i$, it is necessary to identify the start node of an E-path. This can be achieved by starting with the end node, i.e. a node having *Redund* = *true*, and following the E-path backwards to its start node. However, it is more efficient to start with a node which has e available at its exit and which is a predecessor of an E-path suffix, and follow (a part of) availability part of an E-path. The *SA_in/out* data flow is used for this purpose. When e is totally redundant in some b_i, $Eps_in_i = false$ due to the $\neg Av_in_i$ term. Only in such cases the E-path is followed starting with the node having *Redund* = *true*. Note that excepting [13], no other PRE algorithm provides data flow equations to identify save points. The approach in [21] identifies save and insertion points analogously, however it requires isolation analysis to be performed by an additional data flow. The approach in [2] also identifies save and insertion points analogously. It, too, would require an additional data flow problem to perform isolation analysis, else it suffers from a peculiar form of redundant code movement.

Table 1 shows results of the E-path_PRE data flows when applied to the program flow graph of Fig. 1.

Table 1: Solutions of E-path_PRE data flows

Properties	*Nodes*										
Bit 1:$a * b$, 2:$c * d$	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}	b_{11}
Av_in	00	00	00	01	01	01	01	01	11	11	01
Av_out	00	11	01	01	01	01	01	11	11	11	11
Ant_in	01	11	01	00	00	10	10	11	11	11	10
Ant_out	01	00	00	00	10	10	10	11	11	10	00
Eps_in	00	00	00	00	00	00	00	10	00	00	10
Eps_out	00	00	00	00	00	00	00	00	00	00	00
Redund	00	00	00	00	00	00	00	10	10	01	10
Insert	00	00	00	00	00	00	10	00	00	00	00
SA_in	00	00	00	01	00	00	00	01	01	10	00
SA_out	00	01	01	01	00	00	00	11	11	10	00
Save	00	01	01	00	00	00	00	00	00	00	00
$Insert_{4,8} = 10$											

Data flow properties

$Comp_i$	: e is locally available in b_i
$Antloc_i$	: e is locally anticipatable in b_i
$Transp_i$	: b_i does not contain definitions of e's operands
Av_in/out_i	: e is available at entry/exit of b_i
Ant_in/out_i	: e is anticipatable at entry/exit of b_i
Eps_in/out_i	: entry/exit of b_i is in an e-path suffix
$Redund_i$	: occurrence of e in b_i is redundant
$Insert_i$	: insert $t_e \leftarrow e$ in node b_i
$Insert_{ij}$	: insert $t_e \leftarrow e$ along edge (b_i, b_j)
SA_in/out_i	: A Save must be inserted Above the entry/exit of b_i
$Save_i$	: e should be saved in t_e in node b_i

Data flow equations

$$Av_in_i = \Pi_p(Av_out_p) \quad (1)$$
$$Av_out_i = Av_in_i \cdot Transp_i + Comp_i \quad (2)$$
$$Ant_in_i = Ant_out_i \cdot Transp_i + Antloc_i \quad (3)$$
$$Ant_out_i = \Pi_s(Ant_in_s) \quad (4)$$
$$Eps_in_i = \Sigma_p(Av_out_p + Eps_out_p) \cdot Ant_in_i \cdot \neg Av_in_i \quad (5)$$
$$Eps_out_i = Eps_in_i \cdot \neg Antloc_i \quad (6)$$
$$Redund_i = (Eps_in_i + Av_in_i) \cdot Antloc_i \quad (7)$$
$$Insert_i = \neg Av_out_i \cdot \neg Eps_out_i \cdot \Pi_s(Eps_in_s) \quad (8)$$
$$Insert_{ij} = \neg Av_out_i \cdot \neg Eps_out_i \cdot \neg Insert_i \cdot Eps_in_j \quad (9)$$
$$SA_out_i = \Sigma_s(Eps_in_s + Redund_s + SA_in_s) \cdot Av_out_i \quad (10)$$
$$SA_in_i = SA_out_i \cdot \neg Comp_i \quad (11)$$
$$Save_i = SA_out_i \cdot Comp_i \cdot \neg(Redund_i \cdot Transp_i) \quad (12)$$

Figure 3: Data flows of E-path_PRE

3.2.1 Efficiency of data flows

Let (b_i, b_j) be an edge in G. From the theory of bit-vector data flow analysis [19], information flows from exit of node b_i to entry of b_j if a change in the data flow information at exit of b_i causes the corresponding data flow information at the entry of b_j to change. An *information flow path* (ifp) is a sequence of edges along which information can flow during data flow analysis. The bound on the amount of work performed during data flow analysis depends on the length of an information flow path in an obvious way. The number of iterations in a round-robin data flow analysis is bounded by the *width* of a program flow graph for a given data flow problem [10, 19], which is also related to the length of ifp's. For unidirectional data flows, the width is given by the number of back-edges along an information flow path, and is the same as *depth* of a graph [1, 10, 19, 26]. The complexity of data flow analysis and the work done during it is likely to be smaller for data flow problems with shorter ifp's. The observed complexity and work done depend on the paths traced during data flow analysis, i.e. it depends on the ifp's along which information actually flows. For Π and Σ problems, these are paths along which the data flow property is 0 and 1, respectively.

The data flow problems of Fig. 3 have been formulated such that ifp's are short. For example, in Fig. 2 *Eps_in/out* would not be set for any nodes in paths 1, 3 and $4'$. In path 2, *Eps_in/out* would be set only for the nodes in path $2'$. One could have omitted the $\neg Av_in_i$ term from the *Eps_in* equation. However, it would make the ifp's longer. For example, without the $\neg Av_in_i$ term, *Eps_in/out* would be set to 1 for nodes along entire paths 3 and $4'$. It would also be set for the entire anticipatability part of path 2, rather than for $2'$. Hence it would involve more work than setting this property only for nodes in an E-path suffix. The *SA_in/out* data flow similarly sets the property to 1 for parts of an E-path other than the E-path suffix, rather than for the entire E-path.

3.3 Properties of E-path_PRE

Properties of E-path_PRE are presented in Appendix A. Lemma 3 shows that optimization using E-path_PRE is correct, i.e. expression e is deleted in node b_j iff e is available at entry to b_j in the optimized program. Lemma 10 shows that E-path_PRE is optimal, i.e. no other approach using safe insertion of computations leads to a program which contains fewer computations in any path than a program optimized using E-path_PRE. Lemma 11 shows that E-path_PRE does not perform redundant code movement.

3.4 Comparison With Other Approaches to PRE

The PRE approach described in [13], the LCM approach in [21] and its variation reported in [15] all follow the hoisting-followed-by-sinking approach of [4]. Hence these approaches are conceptually more complex and their proofs are more involved than those of E-path_PRE. The approach in [13] identifies insertion, deletion and save points. As mentioned in Section 3.2, [2] would require an additional data flow problem to identify save points. The LCM variation in [15] identifies insertion and deletion points only. It, too, would require an additional data flow problem to identify save points. Thus, E-path_PRE and the approaches in [13], [21] and [15] each require 4 data flow problems.

Approaches in [21] and [2] can handle nodes containing single expressions only. E-path_PRE and the PRE approaches in [4, 13, 15] do not have this restriction. These four approaches perform *edge placement* of computations only when computations cannot be safely placed in nodes to eliminate all eliminatable partial redundancies. Approaches in [12, 21] assume that each *critical edge*, i.e. an edge from a node with > 1 out-edges to a node with > 1 in-edges, is split before performing PRE. [2] performs insertion only along edges. As mentioned earlier, it suffers from a peculiar form of redundant hoisting wherein an expression may be hoisted from a node and inserted into all its in-edges. Lazy code motion [21] performs insertion only at the start of a node. This model of code placement would miss the optimization opportunity in the flow graph of Fig. 4. Hence LCM splits all in-edges of a node having more than one predecessor. This fact leads to more edge-splits than in our approach. [22] uses LCM-like ideas to optimize basic blocks containing more than one statement. In Fig. 4, it correctly moves $a * b$ of node b_2 to node b_1. However, it permits insertion of code at both entry and exit of a basic block and also requires splitting of critical edges. Hence [15] is the closest equivalent approach to E-path_PRE.

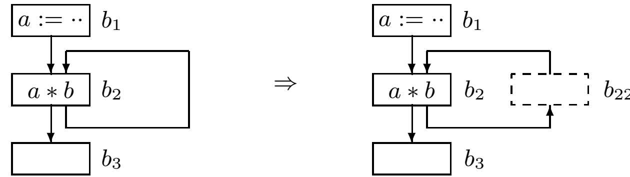

Figure 4: Optimization counter-example for lazy code motion

In this Section we compare E-path_PRE with the approach in [15]. Figure 5 shows the data flow equations reported in [15]. Property *Earliest* indicates the earliest placements of computations of e in $(R_{ant} - R_{av})$. In the *Later* data flow, $Later_in_j$ is *true* if a computation can be placed later along path(s) passing through b_j without sacrificing computational efficiency. Assuming that computation of save points can be performed as in E-path_PRE, [15] and E-path_PRE differ only in the data flows *Later* and *Eps*.

$$
\begin{aligned}
Earliest_{i,j} &= Ant_in_j \cdot \neg Av_out_i, \text{ if } b_i \equiv n_0 \\
&= Ant_in_j \cdot \neg Av_out_i \cdot (\neg Transp_i + \neg Ant_out_i), \text{ otherwise} \\
Later_in_j &= false \text{ if } b_j \equiv n_0 \\
&= \Pi_p \ (Later_{p,j}), \text{ otherwise} \\
Later_{i,j} &= Later_in_i \cdot \neg Antloc_i + Earliest_{i,j} \\
Insert_{i,j} &= Later_{i,j} \cdot \neg Later_in_j \\
Delete_i &= false \text{ if } b_i \equiv n_0 \\
&= Antloc_i \cdot \neg Later_in_i
\end{aligned}
$$

Figure 5: Data flow problems in the variation of LCM reported in [15]

[15] performs conceptual hoisting-followed-by-sinking along a path starting on an edge with *Earliest* = *true* and ending on an edge with *Insert* = *true*. (If no insertions are performed the path ends on a node containing an occurrence of e.) A path segment along which anticipatability is true but the segment does not belong to an E-path is also a conceptual hoisting-followed-by-sinking path in [15]. In Fig. 1, b_5-b_6-b_7 is such a path for $a * b$.

The work done while solving *Later* data flow is higher than in the case of *Eps_in/out* data flow because *Later* traces all paths in G except the conceptual hoisting-followed-by-sinking paths, whereas *Eps_in/out* traces only the suffix of an E-path. This difference is evident from the solution for the sample program of Fig. 1, viz.

Nodes	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}	b_{11}
Eps_in	00	00	00	00	00	00	00	10	00	00	10
Eps_out	00	00	00	00	00	00	00	00	00	00	00
Later_in	00	11	01	00	00	10	10	00	00	00	00

Edges	1-2	1-3	2-4	3-4	4-5	5-6	6-7	7-11	4-8	8-9	9-10	9-8	10-11
Later	11	01	00	00	00	10	10	10	10	00	00	00	00

Since *Eps_in/out* is a Σ problem, initialization at all nodes except n_0 is 0, hence data flow analysis traces those paths along which properties become 1. The data flow comprising *Later* and *Later_in* is a Π problem. Hence initialization at all nodes except n_0 is 1, and paths along which properties become 0 are traced during data flow analysis. From the above table, it is clear that more work has to be done when *Later* is used rather than when *Eps_in/out* is used. For the sample program, the *Eps_in/out* data flow requires 2 iterations in a round-robin iterative solution and involves 2 meet operations in a worklist iterative solution. For the same program, the *Later_in/out* data flow requires 3 iterations and 17 meet operations.

Appendix B reports performance of E-path_PRE. It is found that the *Eps* data flow is 5 times less expensive than *Later* in terms of number of bit vector operations, and 1.75 times less expensive than *Later* in terms of number of iterations. The data flows of E-path_PRE are cumulatively 1.5 times less expensive than the data flows of the LCM variation reported in [15].

4 Concluding Remarks

The difficulties in making PRE easy to understand and practice have been due to the lack of a conceptual basis for PRE. Use of E-path as the basis for PRE leads to an approach which is easy to understand and simple to prove correct. The improvement in solution efficiency is a sheer bonus.

Acknowledgment The author wishes to thank Atul Jawale for the performance studies reported in Appendix B. Atul was partly supported by a grant from Microsoft Research University Relations.

References

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

[2] R. Bodik, R. Gupta, and M. L. Soffa. Complete removal of redundant expressions. *Proceedings of ACM SIGPLAN '98 Conference on PLDI*, pages 1–14, June 1998.

[3] F. Chow et al. A new algorithm for partial redundancy elimination based on SSA form. *Proceedings of the SIGPLAN '97 Conference on PLDI*, pages 273–286, June 1997.

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

[5] D. M. Dhamdhere. Register assignment using code placement techniques. *Computer Languages*, 13(2):75–93, 1988.

[6] D. M. Dhamdhere. A new algorithm for composite hoisting and strength reduction optimisation. *International Journal of Computer Mathematics*, 27:1–14, 1989.

[7] D. M. Dhamdhere. A usually linear algorithm for register assignment using edge placement of load and store instructions. *Computer Languages*, 15(2):83–94, 1990.

[8] D. M. Dhamdhere. Practical adaptation of global optimisation algorithm by morel & renvoise. *ACM Transactions on Programming Languages and Systems*, 13(2):291–294, 1991.

[9] D. M. Dhamdhere and J. R. Isaac. A composite algorithm for strength reduction and code movement. *International Journal of Computers and Information Sciences*, pages 243–273, 1980.

[10] D. M. Dhamdhere and U. P. Khedker. Complexity of bidirectional data flows. *Proceedings of Twentieth annual symposium on POPL*, pages 397–408, 1993.

[11] D. M. Dhamdhere and H. Patil. An elimination algorithm for bi-directional data flow analysis using edge placement technique. *ACM TOPLAS*, 15(2):312–336, 1993.

[12] D. M. Dhamdhere, B. K. Rosen, and F. K. Zadeck. How to analyze large programs efficiently and informatively. *Proceedings of ACM SIGPLAN '92 Conference on PLDI*, pages 212–223, 1992.

[13] V. M. Dhaneshwar and D. M. Dhamdhere. Strength reduction of large expressions. *Journal of Programming Languages*, 3:95–120, 1995.

[14] K. Drechsler and M. P. Stadel. A solution to a problem with morel and renvoise's "global optimization by suppression of partial redundancies". *ACM TOPLAS*, 10(4):635–640, 1988.

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

[16] S. M. Joshi and D. M. Dhamdhere. A composite hoisting-strength reduction transformation for global program optimisation, Parts I and II. *International Journal of Computer Mathematics*, 11(1 & 2):21–24 & 111–126, 1982.

[17] K. Kennedy. Safety of code movement. *International Journal of Computer Mathematics*, 3:112–130, 1972.

[18] R. Kennedy et al. Strength Reduction via SSAPRE. *LNCS*, 1383:144–157, 1998.

[19] U. P. Khedker and D. M. Dhamdhere. A generalised theory of bit vector data flow analysis. *ACM TOPLAS*, 16(5):1472–1511, September 1994.

[20] U. P. Khedker and D. M. Dhamdhere. Bidirectional data flow analysis : Myths and reality. *SIGPLAN Notices*, 34(6):47–57, 1999.

[21] J. Knoop, O. Ruthing, and B. Steffen. Lazy code motion. *ACM SIGPLAN '92 Conference on PLDI*, pages 224–234, June 1992.

[22] J. Knoop, O. Ruthing, and B. Steffen. Optimal code motion: theory and practice. *ACM TOPLAS*, 30(4):1117–1155, 1994.

[23] J. Knoop, O. Ruthing, and B. Steffen. The power of assignment motion. *ACM SIGPLAN '95 Conference on PLDI*, 30(5):233–245, 1995.

[24] R. Lo et al. Register promotion by sparse partial redundancy elimination of loads and stores. *ACM SIGPLAN '98 Conference on PLDI*, pages 26–37, May 1998.

[25] E. Morel and C. Renvoise. Global optimization by suppression of partial rendundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[26] S. S. Muchnick. *Advanced Compiler Design & Implementation.* Morgan Kaufmann, 1997.

[27] V. K. Paleri, Y. N. Srikant, and P. Shankar. A simple algorithm for partial redundancy elimination. *Sigplan Notices*, 33(12):35–43, 1998.

[28] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. *Proceedings of Fifteenth annual symposium on POPL*, pages 12–27, 1988.

[29] O. Ruthing. Bidirectional data flow analysis in code motion: Myth and reality. *LNCS*, 1503:1–16, 1998.

[30] O. Ruthing, J. Knoop, and B. Steffen. Sparse code motion. *Conference record of Principles of Programming Languages*, pages 170–183, 2002.

[31] B. Steffen. Property oriented expansion. *LNCS*, 1145:22–41, 1996.

[32] M. Wolfe. Partial redundancy elimination is not bidirectional. *SIGPLAN Notices*, 34(6):43–46, 1999.

A Properties of E-path_PRE

Let $b_i \dots b_x$-$b_y \dots b_k$ be an E-path such that $b_y \dots b_k$ is an E-path suffix. Thus $b_i \dots b_x$ belongs to the availability part of the E-path. Properties of E-path_PRE can be summarized in the form of following lemmas.

Lemma 1 *$Eps_in_j = true$ iff node b_j lies along an e-path suffix.*

Let $Eps_in_j = true$. From the *Eps_in* equation, $Ant_in_j \cdot \neg Av_in_j = true$. Let node b_j not lie along an E-path suffix. Therefore there is no path $b_i \dots b_j$ in G such that $Av_out_i = true$, $empty((b_i \dots b_j))$ and e is either available or anticipatable at the exit of each node in $[b_i \dots b_j)$. Let $Av_out_p = true$ for some predecessor b_p of b_j. This is a contradiction. Hence let no predecessor of b_j have $Av_out = true$. From Eqs. (5), (6), $Eps_out_p = Eps_in_p = true$ for some predecessor b_p of b_j. Continuing in this manner n_0, the initial node of G, must have $Eps_in = true$. This is a contradiction. Maximal fixed point of *Eps_in/out* ensures the if part.

Lemma 2 *In the optimized program, expression e is available at entry to every node b_j such that $Eps_in_j = true$.*

Let $Eps_in_j = true$. Consider a path $n_0 \dots b_i$-b_j in G such that e is not available at the exit of b_i in the optimized program. Hence $Insert_i = Insert_{ij} = false$. From Eqs. (8), (9), $Eps_out_i = true$, and from Eq. (6) $Eps_in_i = true$. Continuing in this manner $Eps_in_{n_0} = true$. This is a contradiction.

Lemma 3 *An expression e in node b_j is deleted if and only if e is available at entry to b_j in the optimized program.*

From Eq. (7), $Redund_j = true$ implies either $Eps_in_j = true$ or $Av_in_j = true$. Hence proof follows from Lemma 2.

Lemma 4 *$Eps_in_j = true$ iff every path passing through node b_j has a prefix $b_j \dots b_{k*}$ such that $Redund_{k*} = true$ and insertion is not performed in any node or edge along $[b_j \dots b_{k*})$.*

From Lemma 1, $Eps_in_j = true$ iff b_j is situated along the suffix of some E-path $b_i \dots b_k$. $Eps_in_j = true$ implies $Ant_in_j = true$. Hence every path passing through node b_j has a prefix $b_j \dots b_{k*}$ such that b_{k*} contains an occurrence of e and $Ant_out_l \wedge empty(b_l)$ for every node b_l in $[b_j \dots b_{k*})$. From Def. 1, path $b_i \dots b_{k*}$ is also an E-path. Hence $Eps_in_j = true$ iff every path passing through b_j has a prefix $b_j \dots b_{k*}$ which is a part of an E-path suffix. Since $Eps_in = true$ for all nodes in $[b_j \dots b_{k*}]$, insertion is not performed in any node or edge along $[b_j \dots b_{k*})$ (see Eqs. (8), (9)). From Lemma 3, computation of e in b_{k*} will be eliminated.

Lemma 5 *The value of the last occurrence of an expression e is saved in a node b_i if and only if an E-path $b_i \dots b_k$ exists for some b_k, and the last occurrence of e in b_i is not redundant.*

From Eq. (12), $Save_i = true$ implies $\neg(Redund_i \cdot Transp_i)$. Hence the last occurrence of e in b_i is not redundant. $Save_i = true$ also implies $Comp_i = Av_out_i = true$ and $SA_out_i = true$. From Eqs. (10), (11), either $Eps_in_s = true$ or $Redund_s = true$ for a successor node b_s, in which case b_i-b_s is part of an E-path, or $SA_in_s = SA_out_s = true$. Continuing in this manner, either an E-path $b_i \dots b_k$ exists for some b_k, or $SA_in = SA_out = true$ for an exit node of G, which is a contradiction.

Lemma 6 *An expression e is inserted in a node b_h (edge (b_h, b_s)) if and only if every path passing through node b_h (edge (b_h, b_s)) has a prefix $(b_h \dots b_{k*}]$ ($[b_s \dots b_{k*}]$) such that $Redund_{k*} = true$ and insertion is not performed in any node or edge along $[b_s \dots b_{k*})$, where b_s is a successor of b_h.*

$Insert_h = true$ or $Insert_{hs} = true$ implies $Eps_in_s = true$ (Eqs. (8), (9)). Hence proof follows from Lemma 4.

Lemma 7 *An expression is safe and not redundant in the node or edge where it is inserted.*

From Lemma 6, expression e is safe in node b_h or edge (b_h, b_s) where it is inserted. From Eqs. (8), (9), $\neg Av_out_h$.

Lemma 8 *In the optimized program, no path contains more computations than it did in the original program.*

From Lemma 6, when a computation of e is inserted in a node b_h or edge (b_h, b_s), each path passing through node b_s, which is a successor of node b_h, has a prefix $[b_s \dots b_{k*})$ in which no computation of e is inserted, and a computation of e is deleted in b_{k*}. Hence E-path_PRE does not increase the number of computations along any path.

Lemma 9 *A partially redundant computation of e which is not deleted by E-path_PRE cannot be removed by any safe insertion of computations in G.*

Consider a locally anticipatable occurrence of e in some node b_k and a path $b_i \dots b_k$ such that e is locally available at the exit of b_i and $empty((b_i \dots b_k))$. Let $Eps_in_k = false$. From Lemma 1 and Def. 1, path $b_i \dots b_k$ is not an E-path, hence it must contain node(s) with $Ant_in = Av_in = false$. Let b_l be the first such node. It must have a predecessor b_p which does not lie along $b_i \dots b_k$, hence e must be inserted in b_p in order to eliminate it from b_k. However e is not safe in that node.

Lemma 10 *No other safe insertion of computations in a program contains fewer computations along any path than in the optimized program produced using E-path_PRE.*

From Lemmas 6 and 7, optimization is not possible with fewer insertions. From Lemma 9, it is not possible to perform more deletions. Hence the lemma.

Lemma 11 *If b_j is a node in an E-path and b_h is a predecessor of b_j such that node b_h (edge (b_h, b_j)) is selected for insertion of e by E-path_PRE, insertion in node b_j would have been computationally less efficient.*

From Eqs. (8), (9), $Eps_in_j = true$. Hence node b_j exists in some E-path $b_i \dots b_k$. Insertion in b_j would create an E-path $b_i \dots b_j$, hence the inserted computation will be partially redundant in b_j. Insertion in edge (b_h, b_j) removes e from path $b_i \dots b_k$ without inserting it in any path which does not pass through b_j. Insertion of e in b_j instead of edge (b_h, b_j) is therefore computationally less efficient. When e is inserted in node b_h, the same argument applies to each successor node b_s of b_h and edge (b_h, b_s).

Lemma 11 implies that E-path_PRE does not perform redundant code movement.

B Performance of E-path_PRE

Performance studies were conducted on a test-bed using Gensat, a language-independent static analyser interfaced to an LCC front-end, on an X86 architecture. Table 2 presents performance comparison of E-path_PRE and the variation of LCM reported in [15] for some functions in 176.GCC of the Spec CPU2000 Version 1.00 benchmark. We used two standard data flow solvers—a worklist iterative solver and a round-robin (i.e. conventional) iterative solver. Our worklist iterative solver initializes properties as mentioned in Section 3.2 and takes a pass over the program applying data flow functions to nodes and edges to decide which properties should be entered in the initial worklist. Properties are taken off the worklist one-by-one and their effects on predecessor or successor nodes are incorporated. These nodes are put on the worklist if their properties change.

Performance of E-path_PRE and the variation of LCM is summarized in Table 2. The second and third main columns report performance obtained using a worklist iterative and a round-robin iterative solver, respectively. For the worklist iterative solver we count the number of bit vector operations when meet operations are performed and data flow functions are applied. In the *Eps* data flow, each meet operation can be performed in three operations on bit vectors and each function application involves two bit vector operations. The *Later* data flow requires one and three bit vector operations for the meet and the data flow function, respectively. In the second column, we report the number of bit vector operations for individual data flows of E-path_PRE and the LCM variation, and the total number of bit vector operations performed when PRE is performed using E-path_PRE and the LCM variation. Improvements obtained using E-path_PRE are reported. The third column reports the number of iterations required for the *Eps* and *Later* data flows by the round-robin iterative solver.

The last line in Table 2 reports averages over all programs. The % *Imp* column reports percent improvement when E-path_PRE is used instead of LCM. The *Ind* column reports improvements in a one-to-one comparison between *Eps* and *Later*. The average improvement is by 80.4 percent. Thus, *Eps* is 5 times less expensive than *Later*. The *Tot* column reports comparison between the total number of bit vector operations in E-path_PRE and LCM. E-path_PRE provides an improvement of 36.7 percent over LCM, thus it is 1.5 times less expensive. The average number of iterations for *Eps* is 1.6 as against 2.8 for *Later*. The improvement is thus by a factor of 1.75.

Table 2: Performance of E-path_PRE & LCM for 176.GCC from Spec CPU2000

Program				*Number of bit vector operations*									*No. of iterations*	
				Individual data flows					*Total*		*% Imp*			
Name	*n*	*d*	*x*	*Av*	*Ant*	*SA*	*Eps*	*Lat*	*E-p*	*LC*	*Ind*	*Tot*	*Eps*	*Lat*
File bc-emit														
seg_concat	25	30	12	359	232	237	120	684	948	1512	82.5	37.3	2	3
bc_seg_write	59	72	11	740	626	610	325	1616	2301	3592	79.9	35.9	2	3
bc_end_func	33	36	7	355	252	132	132	349	871	1088	62.2	19.9	1	3
File bc-optab														
bc_expand_bin	38	42	29	383	356	303	159	900	1201	1942	82.3	38.2	2	3
File c-common														
type_for_size	92	109	9	653	691	366	366	2055	2076	3765	82.2	44.9	1	2
type_for_mode	123	145	13	878	923	510	490	3631	2801	5942	86.5	52.9	1	2
comb_strings	105	122	25	1707	1139	486	606	1760	3938	5092	65.6	22.7	2	3
File c-decl														
finish_decl	269	341	19	4943	4218	3431	1227	11491	13819	24083	89.3	42.6	2	2
comb_par_dec	100	118	16	1490	1209	485	420	1725	3604	4909	75.6	26.6	1	3
start_decl	98	120	17	1321	910	779	547	2247	3557	5257	75.6	32.3	2	2
File c-iterate														
collect_iterator	132	155	9	1034	1079	709	530	1751	3352	4573	69.7	26.7	1	3
File c-lex														
check_newline	422	518	12	4353	3996	2505	1828	7000	12682	17854	73.9	29.0	1	3
File c-typeck														
build_arr_ref	128	154	25	1659	1263	1206	542	3369	4670	7497	83.9	37.7	1	3
convert_arg	230	286	21	3331	3225	2688	1126	8556	10370	17800	86.8	41.7	2	3
parse_build_bin	157	209	13	3093	1557	767	730	4506	6147	9923	83.8	38.1	1	2
common_type	347	415	30	5880	3459	2406	1490	11630	13235	23375	87.2	43.4	2	3
build_c_cast	184	234	32	2469	1782	1465	819	5605	6535	11321	85.4	42.3	2	3
convert_fo_as	350	436	28	5407	3468	2732	1570	13258	13177	24865	88.2	47.0	2	3
digest_init	251	315	29	3978	2846	1695	1098	6671	9617	15190	83.5	36.7	2	3
File caller-save														
init_callersave	121	143	17	2031	1448	921	533	4087	4933	8487	87.0	41.9	2	3
res_ref_regs	70	85	23	776	643	358	290	1262	2067	3039	77.0	32.0	1	3
										Average	80.4	36.7	1.6	2.8

n : # nodes *d* : # edges *x* : # expressions

Av : *Av_in/out* data flow problem

Ant : *Ant_in/out* data flow problem

SA : *SA_in/out* data flow problem

Eps : *Eps_in/out* data flow problem

Lat : *Later/Later_in* data flow problem

E-p : Total for all data flow problems of E-path_PRE

LC : Total for all data flow problems of LCM

% Imp : % improvement when E-path_PRE is used instead of LCM

Ind : % improvement of *Eps* data flow over *Later* data flow

Tot : % improvement of all data flows of E-path_PRE over all data flows of LCM