Partial Redundancy Elimination in Two Iterative Data Flow Analyses

Reshma Roy[1]
National Institute of Technology, Calicut, India

Sreekala S
National Institute of Technology Calicut, India

Vineeth Paleri
National Institute of Technology Calicut, India

Abstract

Partial Redundancy Elimination (PRE) is a powerful and well-known code optimization. The idea to combine Common Subexpression Elimination and Loop Invariant Code Motion optimizations into a single optimization was originally conceived by Morel and Renvoise. Their algorithm is bidirectional in nature and was not *complete* and *optimal*. Later, Knoop et al. proposed the first complete and optimal algorithm, Lazy Code Motion (LCM), which takes four unidirectional data flow analyses. In a recent paper, Roy et al. proposed an algorithm for PRE that uses three iterative data flow analyses. Here, we propose an efficient algorithm for PRE, which takes only two iterative data flow analyses followed by two computation passes over the program. The algorithm is both *computationally* and *lifetime* optimal. The proposed algorithm computes the information required for performing the transformation in two passes over the program without considering *safety*. The two iterative data flow analyses are required for making the transformation *safe*. The use of well-known data flow analyses, i.e., *available expressions* analysis and *anticipated expressions* analysis, makes the algorithm simple to understand and easy to prove its correctness. The proposed algorithm is more efficient than the existing algorithms since it takes only two iterative data flow analyses. The efficiency of the proposed algorithm is demonstrated by implementing it in LLVM Compiler Infrastructure and comparing the time taken with other selected best-known algorithms.

2012 ACM Subject Classification Software and its engineering → Compilers

Keywords and phrases Static Analysis, Data Flow Analysis, Code Optimization, Partial Redundancy Elimination

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.35

1 Introduction

Partial Redundancy Elimination (PRE) is a code optimization technique used in compiler design to eliminate redundant computations in a program. It focuses on identifying and eliminating computations that are partially redundant, i.e., the computations that occur more than once in a path in the input program. PRE helps reduce the number of instructions executed and can lead to significant performance improvements in a program. Partial redundancy elimination involves the insertion and deletion of computations at appropriate points in the program so that after the transformation, the program contains less than or equal number of occurrences of the original computation in any path. To preserve the semantics of the original program, the insertions of computations corresponding to the transformation must be safe, i.e., the program must not introduce new computations along any path in the original program.

[1] corresponding author

PRE can be either lexical-based or value-based. Lexical-based PRE focuses on eliminating lexically identical expressions on a path, while a value-based PRE eliminates expressions with identical values on a path. In this work, we focus on lexical-based partial redundancy elimination and anticipate that the insights from this work may find its application in the value-based approach as well. Here, we propose an efficient algorithm for partial redundancy elimination using two iterative data flow analyses followed by two computation passes over the program. The data flow analyses used are the well-known classical analyses, i.e., available expressions analysis and anticipated expressions analysis. Unlike the existing works [6, 9, 14, 16], the proposed algorithm requires only two iterative data flow analyses to perform partial redundancy elimination resulting in a significant efficiency gain. The contributions of our work are:

1. A new algorithm for lexical-based PRE, which takes only two iterative data flow analyses compared to at least three analyses in the existing well-known algorithms [6, 9, 14, 16].
2. Correctness proof of the proposed algorithm.
3. An experimental comparison of the proposed algorithm with the selected existing algorithms [9, 16] demonstrating its efficiency and precision.

1.1 Background

Morel and Renvoise, in their seminal work on partial redundancy elimination (PRE) [12], observed that an algorithm for partial redundancy elimination could potentially address both redundancy elimination and the loop invariant code motion simultaneously. Their approach involved four bidirectional data flow analyses. Morel and Renvoise's algorithm did not achieve computational optimality, i.e., it could not eliminate all partially redundant expressions in a program. Subsequently, Dhamdhere [5] improved upon Morel and Renvoise's algorithm by introducing the concept of edge placement, eliminating more partial redundancies. Another challenge in Morel and Renvoise's algorithm was the occurrence of redundant code motion, an issue that Dhamdhere [5] and Drechsler et al. [7] tackled as they implemented various improvements.

Knoop, Ruthing, and Steffen introduced the *lazy code motion algorithm* for partial redundancy elimination (PRE) [9], incorporating four unidirectional data flow analyses. This algorithm stands out for its computational and lifetime optimality, using a hoisting-followed-by-sinking approach. Knoop et al. devised a method to identify the *earliest* and *latest* points for performing the transformation. Another aspect of their algorithm is the preprocessing step, which involves inserting dummy nodes at the edges of nodes with multiple predecessors. Unfortunately, this step leads to unnecessary edge insertions, resulting in overhead. In response to these considerations, Knoop et al. later refined the lazy code motion algorithm to enhance its practical utility [10]. Additionally, Drechsler and Stadel [8] proposed a variant of the lazy code motion algorithm with a primary focus on practical applicability.

In the realm of partial redundancy elimination (PRE), Paleri et al. presented an algorithm utilizing classical data flow analyses, i.e., *availability*, *anticipability*, *partial availability*, and *partial anticipability* [14]. Notably, the introduction of the path concept in their paper enhances the algorithm's comprehensibility. Furthermore, this algorithm is both computationally and lifetime optimal. Originally designed for nodes containing single statements, Paleri et al. later modified their algorithm to nodes containing multiple instructions, such as the standard basic block [15]. In a work akin to the approach by Paleri et al., Dhamdhere introduced the concept of eliminatability paths to address the optimal placement of computations [6]. Like those of prior researchers, Dhamdhere's approach relies on four unidirectional analyses to eliminate partial redundancies. Recent work by Roy et al. [16] describes an algorithm for PRE that is more efficient than the other computationally optimal algorithms available in the literature since it takes only three iterative data flow analyses - *anticipated expressions*, *safe partially available expressions*, and *safe redundancy path* - compared to four analyses taken by the other algorithms.

One limitation of the presented PRE algorithm is its exclusive focus on lexically equivalent expressions. In contrast, a value-based PRE approach can potentially uncover a greater number of redundancies. The value-based method identifies equivalent expressions based on their actual values rather than relying solely on lexical equivalence. This distinction makes it a more powerful optimization technique for effectively eliminating redundant expressions, reported in the literature [4, 11, 13, 17].

2 Notations and Definitions

We found the formal definitions and notations from [14] appropriate for the proposed algorithm. In this section, we give an informal description of the terms used in the algorithm.

2.1 Control Flow graph

We represent a program as a Control Flow Graph (CFG) $G = (N, E, entry, exit)$ where N represents the set of nodes in the graph and E is the set of edges in the graph. We assume that the CFG has two empty basic blocks, an *entry* node which represents the starting point of the graph and an *exit* node to which all exits of the graph go. An *entry* is the unique entry node with no predecessor nodes, and *exit* is the unique exit node without any successor nodes. Each node in the CFG contains at most one statement in the three-address code form. The assignment statement is of the form $x = e$, where x is a variable, and e is an expression built of variables, constants, and operators. The edge from node i to node j is represented as (i, j). The sets of immediate predecessors and immediate successors of a node n are denoted as $pred(n)$ and $succ(n)$, respectively, where $pred(n) = \{m|(m, n) \in E\}$ and $succ(n) = \{m|(n, m) \in E\}$.

2.1.1 Annotated Control Flow Graph

An annotated control flow graph (ACFG) is a CFG annotated with the information obtained from a data flow analysis at every program point in the CFG, i.e., the input and output points of the basic blocks in the CFG.

2.2 Boolean Properties Associated with the Expressions

An expression e is said to be *locally available* from node i, i.e., available at the output point of node i ($\textsc{AvLoc}_i$), if e appears in node i, and the statement in node i does not modify the operands in e. An expression e is said to be *locally anticipated* from node i ($\textsc{AntLoc}_i$), i.e., anticipated at the input point of node i if e appears in node i. An expression e is said to be *transparent* in node i ($\textsc{Transp}_i$), if the execution of the statement in node i does not modify the operands in e.

An expression is *available* at a point if it has been computed along all paths reaching this point with no changes to its operands since the computation. An expression is said to be *anticipated* at a point if every path from this point has a computation of that expression with no changes to its operands in between. An expression e is *partially available* at point p if there is at least one path from *entry* to p which computes e, and after the last such computation before reaching p there is no modification to its operands. An expression e, occurring at a point p, is partially redundant if e is *partially available* at p. An expression e is *partially anticipated* at p if there is at least one path from p to *exit* which computes e with no changes to its operands in between p and the point of occurrence of e. A point is *safe* for insertion of an expression e if the expression is either *available* or *anticipated* at that point. An expression is *safe partially available* at a point p if the expression is *partially available* at p and the path from point k to p is safe, where k is the point from which expression is *partially available* at p. The path formed by connecting adjacent program points where the expression is *safe partially available* is known as *safe partially available path.* An expression is said to be *safe partially anticipated* at a point p if the expression is *partially anticipated* at p and the path from point p to k is *safe*, where k is the point from which e is *partially anticipated* at p. A path is said to be a *safe redundancy path* for an expression if the expression is both *safe partially available* and *safe partially anticipated* at all points on the path.

The notations used for the properties defined in this section, corresponding to an expression e are described below:

Notations	**Data flow properties**
AvLoc_i	: Locally Available at the output point of node i
AntLoc_i	: Locally Anticipated at the input point of node i
Transp_i	: Transparent in node i
$\text{AvIn}_i/\text{AvOut}_i$	: Available at input/output point of node i
$\text{AntIn}_i/\text{AntOut}_i$	: Anticipated at input/output point of node i
$\text{SpavPathIn}_i/\text{SpavPathOut}_i$	: Input/output point of node i is on Safe Partially Available Path
$\text{SredPathIn}_i/\text{SredPathOut}_i$	: Input/output point of node i is on Safe Redundancy Path

The properties for all the nodes in the CFG are expressed in terms of Boolean equations. We used the symbols summation/product (i.e.,$\sum$ / $\prod$) for the confluence operator, + and . for Boolean connectives *or* and *and*, and ¬ for Boolean negation.

3 Basic Concept

We build on the basic concepts from [14]. The basic idea in [14] is briefly outlined below.

Partial Redundancy Elimination consists of two stages: detection and elimination. An expression e at a point p is said to be *partially redundant* if the expression is *partially available* at p. Thus, to detect partially redundant expressions, we require only the information regarding the *partially available expressions.* This information is obtained through *partially available expressions* analysis. In order to eliminate the partially redundant expressions, we require additional information on *partially anticipated expressions*, which is obtained through *partially anticipated expressions* analysis.

The fundamental idea behind partial redundancy elimination is to find *redundancy paths.* To identify the redundancy paths, we first *mark* all the program points where the expression under consideration is both *partially available* and *partially anticipated.* Now, we identify the redundancy paths by connecting the adjacent program points which are *marked.* Partial redundancy elimination is done by *insertions* and *replacements* of expressions

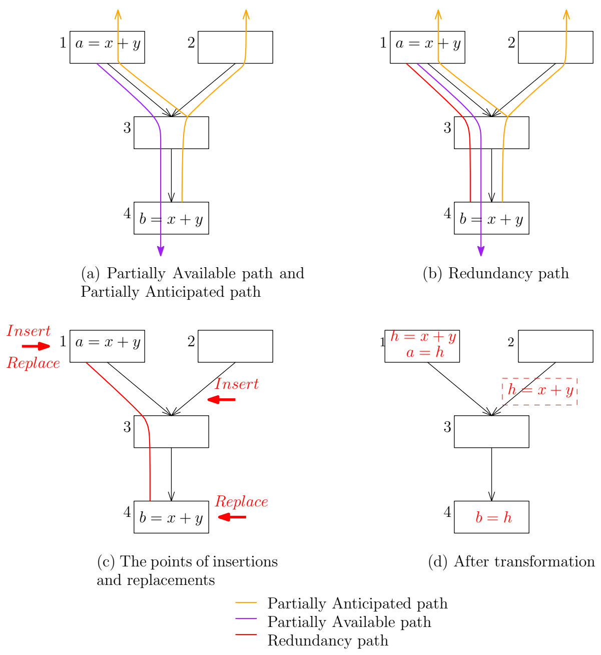


Figure 1 An example for partial redundancy elimination.

at appropriate points in the program [See Fig. 1(c)]. As the initial transformation step, all partially redundant expressions are made totally redundant by inserting the statement $h = e$, where e is the expression of interest, at the edges that enter the junction nodes on the redundancy paths. Now, we insert the statement $h = e$ at the starting points of the redundancy paths. The next step involves the elimination of all the redundant expressions through replacements. The replacement involves the redundant expressions being replaced by the temporary variable h.

We consider the same example in [16] to explain the basic concept. In Fig. 1(a), the purple line represents the path where the expression is *partially available* at all points on the path. Similarly, the orange line denotes the path where the expression is *partially anticipated* at all points on the path. The *redundancy path* is marked in the red line in Fig. 1(b). The insertion points are the input point of node 1 and the edge $(2, 3)$, and the replacement points are nodes 1 and 4, as shown in Fig. 1(c) based on the basic idea explained above. The CFG after the transformation is given in Fig. 1(d).

3.1 The New Approach

As stated above, *redundancy path* is the basic idea behind PRE. We observe an important characteristic of a *redundancy path* corresponding to an expression e. In a *redundancy path*, the first and last nodes contain the expression e [See Fig. 1(c)]. To identify this *redundancy path* for e, we need to visit the nodes in the CFG in a systematic fashion such that e must be *partially available* and *partially anticipated* in the nodes.

A *partially available path* for an expression e starts at a node (say s) containing e in the CFG and moves in the forward direction. We propagate the *partially available* information of e from the node s forward until e is killed or the exit node of the CFG is reached. A *partially anticipated path* for an expression e starts at a node (say t) containing e and moves in the backward direction. We propagate *partially anticipated* information of e from the node t along the *partially available path* computed earlier until e is no longer *partially available*. The path from s to t along which the expression e is both *partially available* and *partially anticipated* forms the *redundancy path*. Thus, the *redundancy path* is obtained using just two computations – not two iterative data flow analyses.

To preserve the semantics of the original program, the insertions of computations corresponding to the transformation done during the PRE algorithm must be safe. We use the notion of *safety* to preserve the semantics of the transformed program. A point p is *safe* for insertion of an expression e if e is *available* or *anticipated* at p. Hence, instead of a simple *redundancy path*, we identify a *safe redundancy path* in the proposed algorithm. The information required to compute *safety* is obtained using two classical data flow analyses: *available expressions* analysis and *anticipated expressions* analysis. After computing *safety* information, *safe redundancy paths* are computed the same way as the computation of *redundancy paths* where propagation must additionally satisfy the *safety* property. Thus, the *safe redundancy path* is identified using just two computations: *safe partially available path* computation and *safe redundancy path* computation, which are detailed in Section 4.

Overall, the algorithm takes two iterative data flow analyses followed by two computation passes over the program.

4 The Proposed Algorithm for PRE

The proposed algorithm consists of two phases: a data flow analysis phase and a computation phase. The first phase has two classical unidirectional data flow analyses: *available expression* analysis and *anticipated expression* analysis. The second phase contains the computations for *safe partially available path* and *safe redundancy path*. The algorithm is presented for a single arbitrary expression e. However, an independent combination of all the expressions in a program will result in a global algorithm for partial redundancy elimination.

A detailed description of the data flow analyses and computations is presented in this section.

4.1 Data Flow Analysis Phase

4.1.1 Available Expression analysis

The available expression analysis (definition provided in Section 2.2) is done in the forward direction of the control flow graph. To solve the forward *available expression* analysis, we need to initialize $\textsc{AvOut}_{entry}$ with the value $\textsc{False}$ because the expression is not available at the output point of the *entry* node. Note that an *entry* node is the first node of a CFG with no instructions in it. We initialize $\textsc{AvOut}_i = \textsc{Top}$ ($\textsc{Top}$ is denoted by $\top$) for all other nodes, as this value will allow the iterative algorithm to converge to the desired value. Note that for a value x, $x \wedge \top = x$. The iterative data flow analysis to compute available expression information is given in Algorithm 1.

Algorithm 1 Iterative data flow analysis to compute available expression information.

```
Input  : Control Flow Graph(CFG), a program expression $e$.
Output : Input CFG annotated with *availability* information at all points for the
         expression $e$.
1  Procedure AvailExpr(CFG, e)
2      $\textsc{AvOut}_{entry} = \textsc{False}$
3      for each node $i \neq entry$ do
4          $\textsc{AvOut}_i = \top$
5      end
6      while changes to any $\textsc{AvOut}$ occur do
7          for each node $i \neq entry$ do
8              $\textsc{AvIn}_i = \prod_{p \in pred(i)} \textsc{AvOut}_p$
9              $\textsc{AvOut}_i = \textsc{AvLoc}_i + \textsc{AvIn}_i.\textsc{Transp}_i$
10         end
11     end
12 end
```

4.1.2 Anticipated Expression analysis

The anticipated expression analysis (definition provided in Section 2.2) is carried out in the backward direction of the control flow graph. To solve the backward *anticipated expression* analysis, we need to initialize $\textsc{AntIn}_{exit}$ with the value $\textsc{False}$ because the expression is not anticipated at the input point of the *exit* node. We initialize $\textsc{AntIn}_i = \top$ for all other nodes, as this value will allow the iterative algorithm to converge to the desired value. The iterative data flow analysis to compute anticipated expression information is given in Algorithm 2.

Algorithm 2 Iterative data flow analysis to compute anticipated expression information.

```
Input  : Control Flow Graph(CFG), a program expression $e$.
Output : Input CFG annotated with *anticipated* information at all points for the
         expression $e$.
1  Procedure AntExpr(CFG, e)
2      $\textsc{AntIn}_{exit} = \textsc{False}$
3      for each node $i \neq exit$ do
4          $\textsc{AntIn}_i = \top$
5      end
6      while changes to any $\textsc{AntIn}$ occur do
7          for each node $i \neq exit$ do
8              $\textsc{AntOut}_i = \prod_{s \in succ(i)} \textsc{AntIn}_s$
9              $\textsc{AntIn}_i = \textsc{AntLoc}_i + \textsc{AntOut}_i.\textsc{Transp}_i$
10         end
11     end
12 end
```

4.2 Computation Phase

The second phase in the proposed algorithm consists of computations for *safe partially available path* and *safe redundancy path.* During this phase, the necessary information is computed by propagating data from specific points along predefined paths. It is important to note that the paths for data propagation are different for the two distinct computations. The worklist method is used to compute both computations.

4.2.1 Worklist

The basic idea of a work list is to maintain a list of nodes to be processed until the list becomes empty. There are three stages in the use of the worklist in the algorithm:

- **Initialization**: The WorkList is initialized with a set of nodes in the CFG containing the expression of interest.
- **Processing WorkList**:
  - GetNode: The node n to be processed next is taken out from the WorkList.
  - Process: Perform the computations on the node n.
  - Update: If there is a change in the value computed for the node n in the processing step, successor or predecessor nodes of n – for *safe partially available path* and *safe redundancy path* computations respectively – are added to the WorkList.
- **Termination:** The algorithm terminates when the WorkList becomes empty, indicating that all the required nodes are processed.

This worklist algorithm propagates the property, i.e., *safe partially available path* or *safe redundancy path*, from specific nodes, with which the worklist is initialized, through the nodes in the control flow graph until the property becomes False. The algorithm is designed in such a way that each point in the CFG is processed only once.

The computations are detailed in the following sections.

4.2.2 Safe Partially Available Path Computation

In the initialization step of the *safe partially available path* computation, the nodes containing the expression of interest are collected and arranged in the reverse post-order sequence of their appearance within the CFG. This order facilitates efficient computation of information in the forward direction, commencing from each expression found within the CFG.

The basic idea is to compute *safe partially available path* for an expression by traversing a *safe path* in the forward direction and marking the points where the expression is also *partially available.* We get a *safe path* by connecting all the adjacent program points that are *safe.* Note that a point p is *safe* for insertion of an expression e if e is either *available* or *anticipated* at p.

The information required to compute *safety* is obtained during the first phase of the algorithm. After collecting *available* expression and *anticipated* expression information in the first phase, instead of computing *safety* as an independent computation, we integrate safety within the safe partially available path computation for efficiency. The computation of *safe partially available path* begins from a node with the expression of interest e and continues forward along the *safe* path until *partial availability* becomes False. Note that *partial availability* becomes False when expression e is *killed.*

Algorithm 3 Computation of safe partially available path for an expression e.

```
Input  : Control Flow Graph annotated with available and anticipated information
         for e.
Output : Input CFG annotated with safe partially available path information for e.
Procedure SafeParAvailExpr(ACFG)
    Create empty WorkList;
    for each node i do      // The order of traversal is reverse post order
        $SpavPathIn_i$ = False
        $SpavPathOut_i$ = False
        $VisitedIn_i$ = False
        $VisitedOut_i$ = False
        if node i contains expression e then
            WorkList.add(i)
    end
    while !WorkList.isEmpty() do
        i = WorkList.remove()
        if !$VisitedOut_i$ then
            $SpavPathOut_i$ = $AvLoc_i$ + $SpavPathIn_i$ . $Transp_i$ [1]
            $VisitedOut_i$ = True
            if change to $SpavPathOut_i$ occur then
                for each node s ∈ succ(i) do
                    if !$VisitedIn_s$ then
                        if $SafeIn_s$ then              // $SafeIn_s$ = $AvIn_s$ + $AntIn_s$
                            $SpavPathIn_s$ = True
                            $VisitedIn_s$ = True
                            WorkList.add(s)
                end
    end
end
```

4.2.3 Safe Redundancy Path Computation

In the initialization phase of the computation for *safe redundancy path*, the nodes containing the expression of interest are stored in the post-order sequence of their appearance within the CFG. This arrangement facilitates the efficient computation of information in a backward direction, commencing from each expression found within the CFG.

The basic idea is to compute *safe redundancy path* for an expression e by traversing a *safe partially available path* in the backward direction and marking the points where the expression is also *partially anticipated*. After computing *safe partially available path*, the *safe redundancy path* computation begins from a node in the initialized work list, and it progresses in a backward direction along the safe partially available path until the *partially anticipated* property becomes False. Note that *partially anticipated* property becomes False when expression e is *killed*.

[1] $\text{AvLoc}_i \implies \text{SafeOut}_i$ and $\text{SpavPathIn}_i . \text{Transp}_i \implies \text{SafeOut}_i$

Algorithm 4 Computation of safe redundancy path for an expression e.

```
Input  : Control Flow Graph annotated with safe partially available path
         information for $e$.
Output : Input CFG annotated with safe redundancy path information for $e$.
Procedure SafeRedPath(ACFG)
    Create empty WorkList;
    for each node $i$ do                    // The order of traversal is post order
        $\text{SRedPathIn}_i$ = False
        $\text{SRedPathOut}_i$ = False
        $\text{VisitedIn}_i$ = False
        $\text{VisitedOut}_i$ = False
        if node $i$ contains expression $e$ then
            WorkList.add(i)
    end
    while !WorkList.isEmpty() do
        i = WorkList.remove()
        if !$\text{VisitedIn}_i$ then
            $\text{SRedPathIn}_i = \text{SpavIn}_i.\ (\text{AntLoc}_i + \text{SRedPathOut}_i.\ \text{Transp}_i)$
            $\text{VisitedIn}_i$ = True
            if change to $\text{SRedPathIn}_i$ occur then
                for each node p ∈ pred(i) do
                    if !$\text{VisitedOut}_p$ then
                        if $\text{SpavPathOut}_p$ then
                            $\text{SRedPathOut}_p$ = True
                            $\text{VisitedOut}_p$ = True
                            WorkList.add(p)
                end
    end
end
```

4.3 The Main Algorithm

The main algorithm for PRE is given in this section. After computing the required information from the two phases of the algorithm given in sections 4.1 and 4.2, the points of transformation are identified. We can divide the conceptual idea behind the algorithm into three stages:

1. Identification of partially redundant computations.
2. Conversion of partially redundant computations into totally redundant computations through insertions of expressions at program points identified. During insertions, we insert an assignment of the form $h = expr$, where h is a new temporary variable.
3. Elimination of all the redundant expressions through replacements at the identified program points. During replacements, we replace some of the original computations of *expr* by h.

We denote the insertion at the entry of node i by Insert_i, insertion on edge (i, j) by $\text{Insert}_{(i,j)}$, and replacement in node i by Replace_i. These terms compute Boolean values, and we use this information to detect the places of insertions and replacements. The proposed algorithm for partial redundancy elimination is given as Algorithm 5. The Transform() function in Algorithm 5 does the necessary transformation using the information computed earlier in the algorithm.

Algorithm 5 Algorithm for Partial Redundancy Elimination.

```
Input  : Control Flow Graph(CFG), a program expression $e$
Output : The input CFG with the partial redundancies of $e$ eliminated.
Procedure PRE (CFG, e)
    AVAILEXPR(CFG, $e$)
    ANTEXPR(CFG, $e$)
    SAFEPARAVAILPATH(ACFG)[2]        // A computation using work list algorithm
    SAFEREDPATH(ACFG)[3]     // A computation using work list algorithm
    for each node i in the CFG do
        $\text{INSERT}_i = \neg\, \text{SREDPATHIN}_i \,.\, \text{SREDPATHOUT}_i$
        $\text{REPLACE}_i = \text{AVLOC}_i.\text{SREDPATHOUT}_i + \text{ANTLOC}_i.\text{SREDPATHIN}_i$
    end
    for each edge (i,j) in CFG do
        $\text{INSERT}_{(i,j)} = \neg\, \text{SREDPATHOUT}_i$
        .
        $\text{SREDPATHIN}_j$
    end
    TRANSFORM(CFG, $\text{INSERT}_{1...n}$, $\text{INSERT}_{(1...n,\ 1...n)}$, $\text{REPLACE}_{1...n}$)        /* $n$ represents the number of nodes in the CFG */
end
```

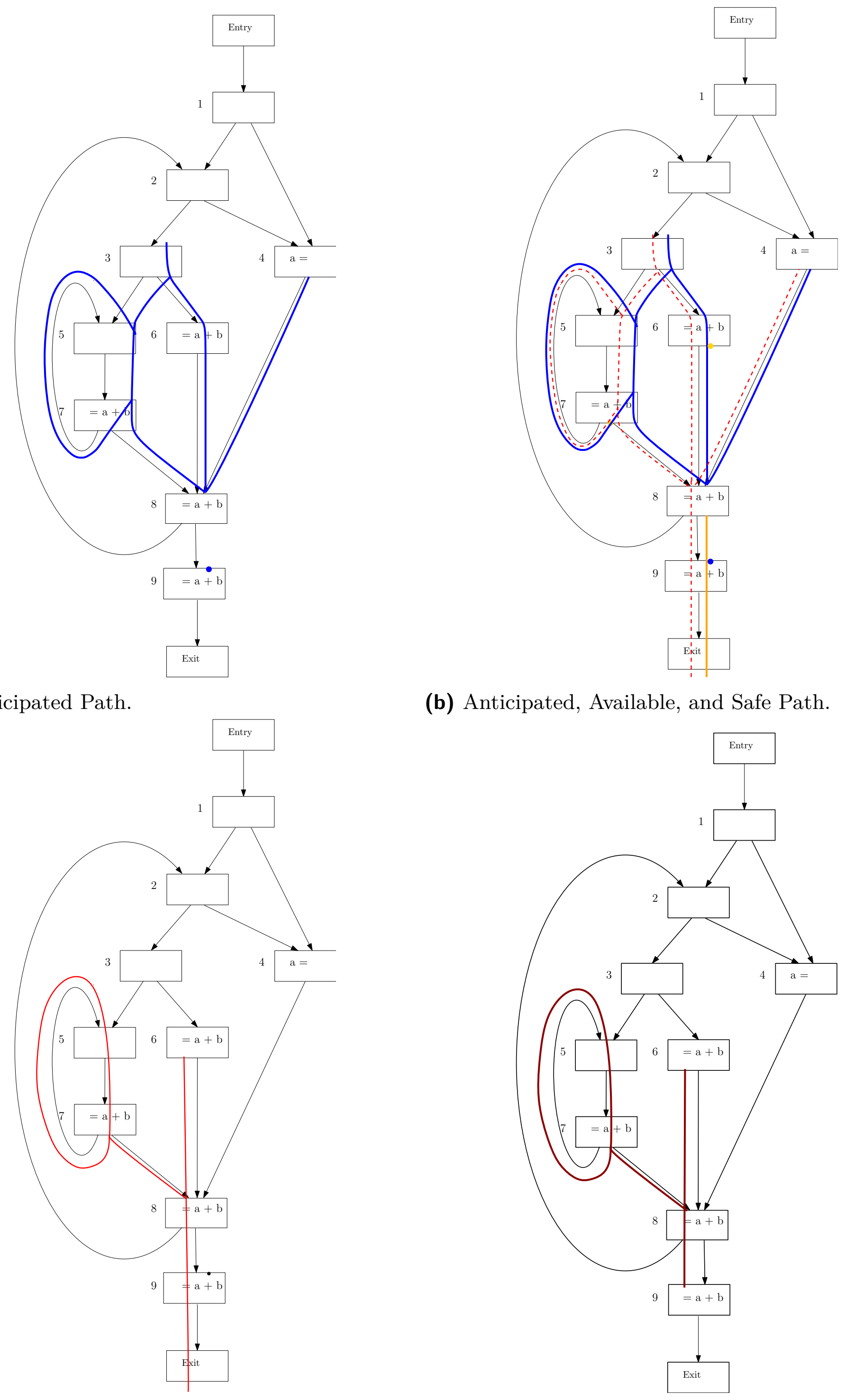
4.4 Example

In Fig. 2, we present an example [14] to illustrate the operation of the proposed algorithm. In Fig. 2(a), the blue line represents the anticipated path. In Fig. 2(b), the orange line shows the available path. The adjacent points which are either blue or orange are joined to form the safe path. The red dotted line in Fig. 2(b) represents safe path. The red line in Fig. 2(c) signifies the safe partially available paths. The brown line in Fig. 2(d) represents the safe redundancy path, which is computed by traversing the safe partially available path in a backward direction and identifying the points where the expression is also partially anticipated. The transformed CFG with insertions and replacements is shown in Fig. 3. The data flow analysis and the transformation information are given in Table 1.

5 Proof of Correctness and Optimality

In this section, we prove the correctness of the analyses performed in the proposed PRE algorithm. In the algorithm, two well-known classical analyses are presented. Therefore, we only provide proof of the correctness of the algorithms for computations in the PRE algorithm. For the proof, as in the algorithm, we consider only one expression e in the input program. Also, our CFG nodes have only a single statement. We assume that a statement of the form $x = x + 1$ is transformed into two statements, $t = x + 1$ and $x = t$, where t is a unique temporary variable.

[2] ACFG with available and anticipated information for the expression e
[3] ACFG with safe partially available path information for the expression e



(a) Anticipated Path.

(b) Anticipated, Available, and Safe Path.

(c) Safe Partially Available Path.

(d) Safe Redundancy Path.

Figure 2 An Example demonstrating PRE by the proposed algorithm.

Table 1 Boolean Properties and Transformations.

Local Boolean Properties			Global Boolean Properties			Insertions and Replacements		
$\textsc{AvLoc}_i$	=	{6, 7, 8, 9}	$\textsc{AntIn}_i$	=	{3, 5, 6, 7, 8, 9}	$\textsc{Insert}_i$	=	{6}
$\textsc{AntLoc}_i$	=	{6, 7, 8, 9}	$\textsc{AntOut}_i$	=	{3, 4, 5, 6, 7}	$\textsc{Insert}_{(i,j)}$	=	{(3, 5), (4, 8)}
			$\textsc{AvIn}_i$	=	{9}	$\textsc{Replace}_i$	=	{6, 7, 8, 9}
			$\textsc{AvOut}_i$	=	{6, 7, 8, 9}			
			$\textsc{SafeIn}_i$	=	{3, 5, 6, 7, 8, 9}			
			$\textsc{SafeOut}_i$	=	{3, 4, 5, 6, 7, 8, 9}			
			$\textsc{SpavPathIn}_i$	=	{5, 7, 8, 9}			
			$\textsc{SpavPathOut}_i$	=	{5, 6, 7, 8, 9}			
			$\textsc{SredPathIn}_i$	=	{5, 7, 8, 9}			
			$\textsc{SredPathOut}_i$	=	{5, 6, 7, 8}			

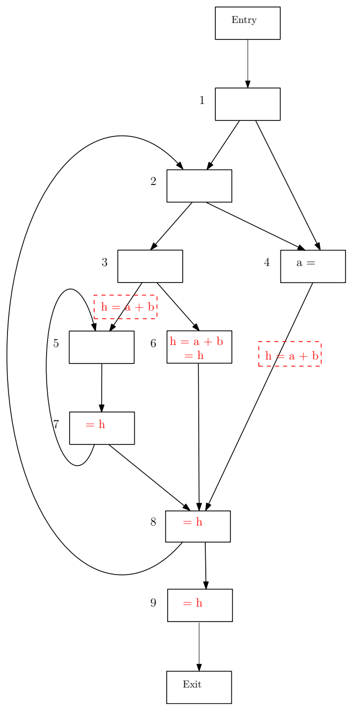


Figure 3 CFG after transformation.

5.1 Correctness of Safe Partially Available Path computation

▶ **Theorem 1** (Correctness). *The computation of the safe partially available path is done correctly.*

Proof. We have to show that every point computed as safe partially available by the safe partially available path computation (Algo.3) is correct. Let N represent the set of nodes in the input CFG.

Axiom 1. {∀ i: i ∈ N: ($\textsc{SpavPathIn}_i$ = $\textsc{False}$) ∧ ($\textsc{SpavPathOut}_i$ = $\textsc{False}$)} at the beginning.

[From initialisation in lines 4-5]

Axiom 2. For an expression e, the input point of node i is on a safe partially available path if the input point of i is safe and the output point of at least one predecessor of node i is on the safe partially available path.
i.e., $\textsc{SpavPathIn}_i = \textsc{SafeIn}_i.\ (\sum_{p \in pred(i)} \textsc{SpavPathOut}_p)$ [By definition, Section 2.2]

Axiom 3. In the algorithm, $\textsc{SpavPathIn}_i$ is set to $\textsc{True}$ for a node i iff
$\{\exists$ p: p $\in$ pred(i): $\textsc{SpavPathOut}_p.\textsc{SafeIn}_i\}$ [Lines 16, 19-20]

▶ **Lemma 2.** *The computation of the safe partially available path at the input point of node i, i.e., $\textsc{SpavPathIn}_i$, is done correctly.*

Proof. Proof is as follows:

Axiom 2 and Axiom 3 $\Rightarrow$ $\textsc{SpavPathIn}_i$ is set to $\textsc{True}$ at the input point of node i if and only if safe partial availability is true – **(1)**

(1) and Axiom 1 $\Rightarrow$ The input point of node i which is not safe partially available remains $\textsc{False}$ – **(2)**

(1) and (2) $\Rightarrow$ Lemma 2 ◀

Axiom 4. For an expression e, the output point of node i is on a safe partially available path, if e is locally available or the input point of node i is on safe partially available path and e is transparent in i. i.e., $\textsc{SpavPathOut}_i = \textsc{Avloc}_i + \textsc{SpavPathIn}_i.\textsc{Transp}_i$. [By definition, Section 2.2]

▶ **Lemma 3.** *The computation of the safe partially available path at the output point of node i, i.e.,$\textsc{SpavPathOut}_i$, is done correctly.*

Proof. $\textsc{SpavPathOut}_i$ is changed only for the nodes that are added to the work list. Therefore, we consider the nodes that are added to the work list. If a node i is added to the work list, then either of the following cases holds.

Case 1. Node i contains expression e. [Line 9]

$\Rightarrow$ $\textsc{AvLoc}_i$ – **(3)**
[Note that in our CFG, a block has only one instruction. Also, an instruction of the form $x = x + 1$ is transformed into two statements, $t = x + 1$ and $x = t$.]

$\Rightarrow$ $\textsc{SpavPathOut}_i$ [By Axiom 4] – **(4)**

Case 2. Node i does not contain the expression e (i.e. $\textsc{AvLoc}_i$ is $\textsc{False}$) and $\textsc{SpavPathIn}_i$ is $\textsc{True}$. [Lines 20, 22] – **(5)**

We need to prove that, under the condition (5), $\textsc{SpavPathOut}_i$ is set to $\textsc{True}$ if and only if $\textsc{Transp}_i$ is $\textsc{True}$ (as given in line 14).

$$\begin{aligned}
\textsc{SpavPathOut}_i &\equiv \textsc{AvLoc}_i + \textsc{SpavPathIn}_i.\textsc{Transp}_i && \text{[By Axiom 4]}\\
&\equiv \textsc{False} + \textsc{SpavPathIn}_i.\textsc{Transp}_i && [\textsc{AvLoc}_i = \textsc{False}, \text{From (5)}]\\
&\equiv \textsc{SpavPathIn}_i.\textsc{Transp}_i && [\textsc{False} + \textsc{p} \equiv \textsc{p}]\\
&\equiv \textsc{True}.\textsc{Transp}_i && [\textsc{SpavPathIn}_i = \textsc{True}, \text{From (5)}]\\
&\equiv \textsc{Transp}_i && [\textsc{True}\ .\ \textsc{p} \equiv \textsc{p}]
\end{aligned}$$

i.e., $\textsc{SpavPathOut}_i$ is $\textsc{True}$ iff node i is transparent.

Hence, $\textsc{SpavPathOut}_i$ is set to $\textsc{True}$ correctly in case 2. – **(6)**

(4) and (6)	$\Rightarrow$	SpavPathOut$_i$ is set to True at the output point of node i if and only if safe partial availability is true. – **(7)**
(7) and Axiom 1	$\Rightarrow$	The output point of node i which is not safe partially available remains False. – **(8)**
(7) and (8)	$\Rightarrow$	Lemma 3 ◀

Lemma 2 and Lemma 3 => Theorem 1 ◀

▶ **Theorem 4** (Completeness). *The computation of the safe partially available path identifies all points that are safe partially available.*

Proof. We take three stages in the computation to prove the completeness:

- *Starting node of a safe partially available path*: A safe partially available path begins at the output point of a node containing the expression e. A node i containing the expression e is added to the work list in lines 8-9. The node i is then taken out from the work list (line 12) and safe partial availability information at the output point of node i is computed correctly in line 14.
- *Propagation of information:* The safe partial availability information at the output point of node i is then propagated to the input point of each of the successor nodes, say j, if the input point of node j is safe (lines 17-20), and those successor nodes are added to the work list (line 22). Each of these successor nodes is later taken out from the work list (line 12), and the information is further propagated from the input point to the output point of node j if node j is transparent (line 14).
- *End node of a safe partially available path:* The propagation ends under two conditions:
  (i) The propagation from the input point to the output point of node i ends if e is killed in i.
  (ii) The propagation from the output point of node i to the input point of its successor node j ends if input point of node j is not safe.

  Hence, all possible safe partially available paths starting from a node i are computed correctly during this process.

This process of propagation of safe partial availability information is performed from each node containing e in the given input program. Hence, the computation identifies all points on the safe partially available path. ◀

▶ **Theorem 5** (Termination). *Safe partially available path computation terminates.*

Proof. The algorithm terminates when the work list is empty (line 11). Initially, the work list contains nodes with the expression e from the input program (lines 8-9). After that, a node i is added to the work list if there is a change of value in SpavPathOut$_p$ where p $\in$ pred(i) (lines 16, 22). The value in SpavPathOut$_i$ of a node i can change from the initialized value False (line 5) to True at most once (line 14), owing to the fact that once the value becomes True, it remains True. Hence, the number of nodes added to the work list after initialization equals the number of value changes for SpavPathOut. If the total number of nodes in the CFG is N, then there can be at most N number of value changes. Since the nodes from the work list are removed (line 12) for computing SpavPathOut, and the number of node additions is at most N, eventually the work list becomes empty. Hence, the algorithm terminates. ◀

5.2 Correctness of Safe Redundancy Path computation

▶ **Theorem 6** (Correctness and Completeness). *The computation of the safe redundancy path is correct and complete.*

The line of reasoning is similar to the reasoning given for *safe partially available path*, except for the fact that the propagation in this case is in the backward direction and necessary changes accordingly. Hence, the formal proof is avoided here.

5.3 Optimality of Transformation

▶ **Theorem 7.** *The transformation in the proposed PRE algorithm is computationally and lifetime optimal.*

The proposed algorithm is based on the idea of the safe redundancy path in [14]. The transformation done on the safe redundancy path is proved to be both computationally and lifetime optimal in [14].

6 Experimental Results

In this section, we perform an experimental evaluation to compare the proposed algorithm with existing ones. For comparison, we consider two aspects: the number of redundancies detected (i.e., precision) and the running time of the algorithms. We have selected two of the existing algorithms which are computationally and lifetime optimal for comparison. The algorithms chosen for this comparison are: LCM [9], the well-known PRE algorithm which takes four analyses, and PRE-3 [16], which takes three analyses.

In the proposed work, the algorithm is designed for an arbitrary expression e. For implementation, we employ bit-vector representation to extend the algorithm to all the n expressions within the program. At a program point in the CFG, each property (e.g., $\textsc{SpavPathOut}_i$ in Algo. 3) is represented by a bit vector. Each bit in the bit vector corresponds to an expression where $\textsc{True}$ means the property is true for the expression, while $\textsc{False}$ means the property is false.

To illustrate how a bit vector facilitates parallel computation of all n expressions within the program, let's examine the computation $\textsc{SpavPathOut}_i = \textsc{AvLoc}_i + \textsc{SpavPathIn}_i.\textsc{Transp}_i$ in Algorithm 3. Consider the computation $\textsc{SpavPathIn}_i.\ \textsc{Transp}_i$, where $\textsc{SpavPathIn}_i$ and $\textsc{Transp}_i$ are bit-vectors representing the information for n expressions. An AND operation between the bit-vectors $\textsc{SpavPathIn}_i$ and $\textsc{Transp}_i$ results in the bit-vector representing the property $\textsc{SpavPathOut}_i$ for all the n expressions at the output point of node i.

We used LLVM compiler infrastructure [1, 2] for our implementation. The results were obtained on a machine with a 1.8 GHz Intel Core i5 processor having 8 GB RAM for selected programs from the SPEC CPU2006 benchmark suite [3]. The analyses are intraprocedural. The algorithm is implemented for demonstrating its *completeness* and *efficiency*. Accordingly, we have decided to consider a subset of instructions i.e., instructions involving signed and unsigned integer arithmetic operators (+, -, $*$, $\div$, %) to simplify the implementations. The LLVM IR instructions considered are *add*, *sub*, *mul*, *udiv*, *sdiv*, *urem* and *srem* as well as the *load* and *store* instructions of normal variables which includes both local and global variables. For other instructions, we made conservative assumptions. For example, consider a statement with pointer assignment, $*p = \ldots$. This statement may change the value of normal variables of the program. So, we made a conservative assumption that all the variables are killed at the output point of such an instruction.

For our experiment, we begin with some preprocessing steps. We employ the *-instnamer* pass in LLVM to assign names to any unnamed values within the LLVM IR code. This is necessary as these values are not accessible through the *getName()* method we have used. We wanted only the instructions that can be reached from the entry node. To achieve this, we execute the *-unreachableblockelim* pass provided by LLVM. For Algorithm 3 and Algorithm 4, the worklist is implemented with the *InstructionWorkList* in llvm. This *InstructionWorkList* is implemented using a stack in llvm.

6.1 Efficiency

In this section, we compare the execution time of the proposed algorithm against the other two chosen algorithms: the LCM algorithm developed by Knoop et al. and PRE-3 by Roy et al. The algorithms were implemented as passes in the LLVM compiler and were run on selected programs from the SPEC CPU2006 benchmark suite [3] using the *-time-pass* optimizer tool of LLVM to measure execution time. The time taken for analyses by the CPU is measured where the reported time is the sum of the CPU time in user mode and the CPU time in system mode. We execute each benchmark program ten times, employing the *time-pass* functionality. We then calculate the average time from these ten runs. The time taken for analysis by each algorithm is then presented in seconds.

In Table 2, the second column displays the overall count of LLVM IR instructions within each benchmark program. The third column provides the total count of expressions considered, adhering to our conservative assumptions. The subsequent columns provide the time taken by each algorithm under consideration. The final row of the table presents the average time taken by each algorithm, taking into account all the selected benchmark programs.

The proposed algorithm performs better since it takes only two iterative data flow analyses compared to four by LCM and three by PRE-3. The proposed algorithm achieves 51% and 21% reduction in time over LCM and PRE-3, respectively, for the selected set of benchmark programs. The experimental results demonstrate that the proposed algorithm is more efficient in terms of the time taken for analysis compared to the other algorithms.

6.2 Precision

This section looks at the precision of the chosen algorithms, specifically focusing on their completeness in identifying redundant expressions. For the LCM algorithm, we record the count of insertions identified at nodes, the count of insertions specifically at the dummy nodes generated during preprocessing, replacement counts, and the total number of redundant expressions identified. In the case of PRE-3 and the proposed algorithm, we present the count of node insertions, edge insertions, replacements, and the total number of redundant expressions identified. Table 3 provides the information computed during the process. The total number of node insertions for LCM is displayed in column 2, which includes dummy nodes. Column 3 displays the number of dummy nodes created by the algorithm for LCM and used for insertions. Dummy node insertions in LCM are the *edge insertions* in PRE-3 and the Proposed Algorithm. The table demonstrates that the proposed algorithm detects the same number of redundancies as LCM and PRE-3, affirming the completeness of the algorithm. Moreover, upon examining the data in the table, it becomes evident that the identified points of insertions, replacements, and edge insertions are the same for all three algorithms.

Table 2 Comparison of Efficiency.

Benchmark Programs	**No.of instru-ctions**	**Expressions considered**	**Time (Seconds)**		
			LCM	**PRE-3**	**Proposed Algorithm**
astar	11887	260	1.06	0.69	0.63
bzip2	27346	694	33.99	24.87	21.62
gcc	339578	1511	450.52	222.95	170.30
gromacs	185285	3605	40.56	27.71	23.96
h264ref	188827	6302	285.44	218.84	161.06
hmmer	90070	2077	19.47	13.22	11.60
lbm	6155	1131	0.24	0.21	0.14
mcf	3917	90	0.24	0.18	0.16
povray	232142	2049	54.67	36.73	31.87
sjeng	32215	1460	22.60	13.51	12.79
soplex	133448	996	12.81	10.13	9.68
sphinx	47367	824	6.47	4.68	4.13
Average running time			**77.33**	**47.81**	**37.32**

Table 3 Comparison of Precision.

Benchmark Programs	**LCM**				**PRE-3**				**Proposed Algorithm**			
	Insertions (nodes in original CFG + dummy nodes added)	Insertions (dummy nodes)	Replace ments	Redundant expressions detected	Insertions (node)	Insertions (edge)	Replace ments	Redundant expressions detected	Insertions (node)	Insertions (edge)	Replace ments	Redundant expressions detected
astar	13	3	34	24	10	3	34	24	10	3	34	24
bzip2	39	3	82	46	36	3	82	46	36	3	82	46
gcc	61	16	135	90	45	16	135	90	45	16	135	90
gromacs	262	98	532	368	164	98	532	368	164	98	532	368
h264ref	686	99	2057	1470	587	99	2057	1470	587	99	2057	1470
hmmer	220	51	580	411	169	51	580	411	169	51	580	411
lbm	132	0	919	787	132	0	919	787	132	0	919	787
mcf	10	1	45	36	9	1	45	36	9	1	45	36
povray	136	32	317	213	104	32	317	213	104	32	317	213
sjeng	161	13	363	215	148	13	363	215	148	13	363	215
soplex	48	14	70	36	34	14	70	36	34	14	70	36
sphinx	38	10	66	38	28	10	66	38	28	10	66	38

7 Conclusion

In this paper, we presented a novel algorithm for lexical-based partial redundancy elimination. The proposed algorithm takes two iterative data flow analyses followed by two computation passes over the program to perform the transformation. The use of well-known data flow analyses, i.e., *available expressions* analysis and *anticipated expressions* analysis, makes it easy to comprehend the algorithm and prove its correctness. We have provided the proof for the correctness of the algorithm. The algorithm is more efficient compared to other computationally and lifetime optimal algorithms in the literature, as it takes only two iterative data flow analyses, in contrast to at least three analyses required by other methods. The algorithm is both computationally and lifetime optimal. To substantiate these claims, we implemented the algorithm using the LLVM Compiler Infrastructure and compared the number of redundant expressions detected and the time taken for analyses against the existing algorithms. The results from the experiments conducted demonstrate that the proposed algorithm detects the same number of redundant expressions and performs significantly better compared to the existing well-known algorithms considered. Although our algorithm is lexical-based, we believe that its fundamental principles hold significant potential for guiding the transition to a value-based approach, which could ultimately result in an efficient value-based PRE.

References

1 The LLVM Compiler Infrastructure Project. `http://llvm.org/`. Accessed on 03/07/2021.

2 LLVM programmer's manual. `https://llvm.org/docs/ProgrammersManual.html`. Accessed on 20-08-2021.

3 The SPEC CPU2006 benchmark suit. `https://www.spec.org/cpu2006/`, 2006. Accessed on 10-01-2022.

4 Rastisalv Bodík and Sadun Anik. Path-sensitive value-flow analysis. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 237–251, New York, NY, USA, 1998. Association for Computing Machinery. `doi:10.1145/268946.268966`.

5 D. M. Dhamdhere. A fast algorithm for code movement optimisation. *ACM SIGPLAN Not.*, 23(10):172–180, October 1988. `doi:10.1145/51607.51621`.

6 Dhananjay M. Dhamdhere. E_path−PRE: Partial redundancy elimination made easy. *ACM SIGPLAN Not.*, 37(8):53–65, August 2002. `doi:10.1145/596992.597004`.

7 KarlHeinz Drechsler and Manfred P. Stadel. A solution to a problem with Morel and Renvoise's global optimization by suppression of partial redundancies. *ACM Trans. Program. Lang. Syst.*, 10(4):635–640, October 1988. `doi:10.1145/48022.214509`.

8 Karl-Heinz Drechsler and Manfred P. Stadel. A variation of Knoop, Rüthing, and Steffen's lazy code motion. *ACM SIGPLAN Not.*, 28(5):29–38, May 1993. `doi:10.1145/152819.152823`.

9 Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy code motion. *ACM SIGPLAN Not.*, 27(7):224–234, July 1992. `doi:10.1145/143103.143136`.

10 Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Optimal code motion: Theory and practice. *ACM Trans. Program. Lang. Syst.*, 16(4):1117–1155, July 1994. `doi:10.1145/183432.183443`.

11 Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Expansion-based removal of semantic partial redundancies. In Stefan Jähnichen, editor, *Compiler Construction*, pages 91–106, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

12 E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Commun. ACM*, 22(2):96–103, February 1979. `doi:10.1145/359060.359069`.

13 Rei Odaira and Kei Hiraki. Partial value number redundancy elimination. In Rudolf Eigenmann, Zhiyuan Li, and Samuel P. Midkiff, editors, *Languages and Compilers for High Performance Computing*, pages 409–423, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

14 Vineeth Kumar Paleri, Y. N. Srikant, and Priti Shankar. A simple algorithm for partial redundancy elimination. *ACM SIGPLAN Not.*, 33(12):35–43, December 1998. `doi:10.1145/307824.307851`.

15 Vineeth Kumar Paleri, Y. N. Srikant, and Priti Shankar. Partial redundancy elimination: a simple, pragmatic, and provably correct algorithm. *Sci. Comput. Program.*, 48(1):1–20, 2003. `doi:10.1016/S0167-6423(02)00083-7`.

16 Reshma Roy and Vineeth Paleri. Lexical-based partial redundancy elimination: An optimal algorithm with improved efficiency. *Journal of Computer Languages*, 75:101204, 2023. `doi:10.1016/j.cola.2023.101204`.

17 Thomas VanDrunen and Antony L. Hosking. Value-based partial redundancy elimination. In Evelyn Duesterwald, editor, *Compiler Construction*, pages 167–184, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.