Bidirectional Data Flow Analysis : Myths and Reality

Uday P. Khedker
(uday@cs.unipune.ernet.in)
Department of Computer Science
University of Pune, Pune 411 007, India.

Dhananjay M. Dhamdhere
(dmd@cse.iitb.ernet.in)
Department of Computer Science & Engg.
Indian Institute of Technology, Bombay 400 076, India.

Abstract

Research in bidirectional data flow analysis seems to have come to a halt due to an impression that the case for bidirectional data flow analysis has been considerably weakened by a plethora of investigations based on decomposability of known bidirectional placement algorithms into a sequence of purely unidirectional components. This paper shows that the approach of decomposability is not general enough in that it derives its power from the simplifying graph transformation of edge-splitting and the favourable nature of flows in partial redundancy elimination (PRE). This follows from the fact that in the absence of edge-splitting, PRE cannot be performed using a sequence of cascaded unidirectional flows. Further, edge-splitting inherently converts data flows involved in PRE into unidirectional flows. In our opinion, this obviates the need of an alternative formulation. We also show that edge-splitting cannot convert data flows involved in "truly" bidirectional data flow problems into unidirectional flows. Thus not every bidirectional data flow problem can be converted into unidirectional flows. Besides, we argue that the premise that bidirectional analysis is more complex than unidirectional analysis, is invalid. We also mention some issues in bidirectional data flow analysis which need to be investigated.

1 Background

Language processing tools have employed various combinations of forward and backward data flow dependencies for a variety of purposes viz.

- Partial Redundancy Elimination (PRE) [5, 9, 22, 44, 47, 54] and related optimizations [6, 16, 39, 53].
- Code Hoisting and Strength Reduction [15, 17, 21, 26, 29, 30, 36].
- Live Range Characterisation and Register Assignment [14, 41].
- Dead Code Elimination [4, 38].
- Communication Placement [25, 27].
- Shrink Wrapping of Procedure Calls [8].
- Redundant Array Bound Checks Elimination [40].
- Compilation for Distributed Memory [1].

Data flow analysis for some of the above optimizations involves a *sequence of separate flows* in forward and backward directions while for others it involves *simultaneous flows* in forward and backward directions. Analysis of the former kind has been formulated in terms of a sequence of purely unidirectional data flow problems while that of the latter kind has been formulated in terms of bidirectional data flow problems. It was believed for a long time that bidirectional data flow analysis is more complex than unidirectional analysis.

Contributions made by the above investigations fall in one or more of the following categories :

Contribution 1 : Reducing the complexity of data flow analysis by avoiding bidirectional dependencies.

Contribution 2 : Enhancing the effectiveness of optimization by eliminating more partial redundancies.

Contribution 3 : Performing optimal code placement thereby avoiding redundant hoisting.

Contribution 4 : Extending the PRE framework, originally conceived in the context of optimization of expressions, to other domains.

The first contribution lies in the domain of the theory and practice of *performing data flow analysis* while the last three lie in the domain of *semantics of data flow analysis*. Since the former category concerns "how" and the latter concerns "what", ideally they should be orthogonal. It should be possible to discuss the process of performing data flow analysis independently of the semantic information captured by data flow analysis. At the same time, it should be possible to argue about semantic information captured by data flow analysis independently of the method used for performing analysis.

2 Bidirectional Data Flow Analysis : Myths and Reality

Ever since optimizations involving bidirectional dependencies were invented, there have been many attempts at explaining bidirectional flows at a theoretical level with a goal of discovering complexity bounds and reducing the complexity of bidirectional data flow analysis. In this regard, the discovery that bidirectional flows can be converted into approximately equivalent unidirectional flow(s) seems to have been made simultaneously and independently in [13] and [22]. Similar approaches were later used in [6, 19–21, 23, 26, 27, 35–39, 47]. The common thread through all these investigations has been performing bidirectional analysis (or its equivalent) using a sequence of unidirectional flows, hence they seem to lead to the impression that :

> Bidirectional analysis is more complex than unidirectional analysis and since there are ways of avoiding it for most of the known practical problems, the case for bidirectional analysis is considerably weakened.

This impression came to our notice first in the form of referee's remarks on one of our submissions coupled with the absence of any reported investigation involving simultaneous flows in forward and backward directions. Similar opinions were later echoed in [45]. Excerpts :

> "*Bidirectional* problems require forward and backward propagation at the same time and are significantly more complicated to formulate, understand, and solve than one-directional problems. Happily, in optimization, bidirectional problems are rare. The most important instance is the classic formulation of partial redundancy elimination, mentioned in Section 13.3, and even it has been superseded by the more modern version presented there, which uses only unidirectional analyses." . . . (Page 229 in [45])
>
> "Partial redundancy elimination originated by the work of Morel and Renvoise (1979) . . . As formulated by them, the intraprocedural version involves performing a complicated bidirectional data flow analysis, but as we shall see, the modern version that we discuss here avoids this." . . . (Page 407 in [45])
>
> "A key point in the algorithm is that it can be much more effective if the critical edges in the flow graph have been split before the analysis is performed."
>
> [Page 407 in [45]]

The reality, however, is that :

1. *Bidirectional analysis is not more complex than unidirectional analysis* [33].
2. *The alternative formulations cited earlier derive their ability of performing bidirectional analysis from the graph transformation of edge-splitting rather than their characterizations of data flows.*

A careful examination reveals that the two orthogonal domains of the process of analysis and semantics of analysis have been mixed up leading to avoidable confusion. The real contributions of the investigations cited above lie in semantics of PRE. Their contributions to process of performing analysis are adhoc since their ability to perform bidirectional analysis is incidental — it is enabled by edge-splitting rather than by their characterizations of data flows. Further, the fact that edge-splitting helps in PRE is incidental — it is enabled by favourable flows in PRE.

We elaborate on these points in the next two sections. This discussion trivially follows from [18, 32, 33].

2.1 Complexity of Bidirectional Data Flow Analysis

The misconception about the complexity is an outcome of some initial explorations in bidirectional analysis which were adhoc in nature. These initial forays [13, 19, 20, 35] seem to have received a much wider attention than the Generalized Theory of Bit Vector Data Flow Analysis [18, 32, 33] which has shown that bidirectional data flow problems are inherently no more complex than unidirectional problems, in terms of both conceptual and computational complexity.

- *Conceptual Complexity (of the process of performing analysis)*
  The Generalized Theory of Bit Vector Data Flow Analysis subsumes the classical theory of data flow analysis [2, 24, 28, 31, 34, 42, 48] as a special case. This is achieved by proposing a more general concept of flow of information which is uniformly applicable to unidirectional and bidirectional data flows. Thus, flows arising due to bidirectional data flow equations can be understood much the same way as the flows arising due to unidirectional data flow equations.
  The theory of "multi-source" information flows [43] is a possible alternative for explaining bidirectional flows; they have formulated PRE as an example. However, they visualise information as a tuple in which each element is essentially independent until analysis is over. The values of these elements are combined to yield the final value at the end and *not during* analysis. As a consequence, their formulation of PRE is neither distributive nor 2-bounded whereas the Generalized Theory has shown that the original formulation is both distributive as well as 2-bounded.

- *Conceptual Complexity (of understanding semantics of data flows)*
  This complexity remains much the same since semantics are specific to the analysis being performed and as such would be different for different applications of data flow analysis. It is difficult to visualize a general theory of data flow semantics capable of explaining semantics of *all* conceivable applications of data flow analysis. [35] is a good attempt for PRE except that their theory does not hold unless edge-splitting is performed. The only hope in this regard, perhaps is abstract interpretation [10–12, 51].

- *Computational Complexity*
  The Generalized Theory also shows that the worst case complexity of worklist method of data flow analysis is the same for both unidirectional and bidirectional data flows *even if the graph transformation of edge-splitting is not performed*. However, in terms of the average performance bidirectional analysis is likely to be less efficient than unidirectional analysis; nevertheless *this inefficiency is not of the order of magnitude as was believed earlier*. Further, if the brute force method of round robin iterative data flow analysis is used, bidirectional data flow problems may require more iterations than unidirectional data flow problems. [18] provides a bound on the number of iterations for the round robin method of bidirectional data flow analysis.
  Elimination methods [3, 24, 50, 52, 56] depend upon the structural properties of the control flow graphs. Since the information flow in bidirectional data flow analysis does not follow graph structure, these methods are not directly applicable to bidirectional analysis. However, with the help of the concept of *Cyclic Information Flow Graphs* [55] all these methods may be used for bidirectional flows.

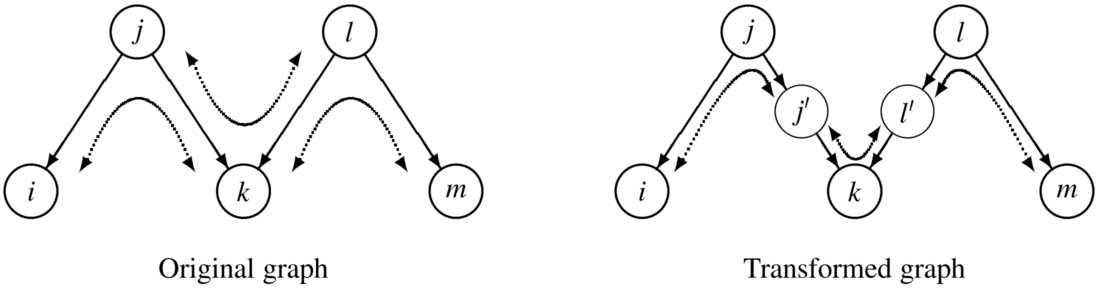


Figure 1: Edge splitting

2.2 Though Shall Split Critical Edges, Else ...

2.2.1 What is edge-splitting?

An edge which runs from a *branch node* (i.e. a node with more than one successor viz. nodes j and l in figure 1) to a *join node* (i.e. a node with more than one predecessor viz. node k) is called a *critical edge* [49]. [13, 19, 21, 22] reported that the solution complexity of PRE is reduced if we consider placement of computations along edges in a program flow graph. [20, 35] show that it is enough to consider *critical edges* for placement of computations and proposed that each critical edge should be split by inserting a new node. This observation can be explained using the Generalized Theory as follows :

> In PRE framework, backward node flow exists and forward node flow does not exist. Thus data flow properties PPIN_u for some node u depend on the data flow properties PPOUT_u but not vice-versa [44]. At the same time for an edge (u,v), PPIN_v depends on PPOUT_u and vice-versa. For example, PPIN_k depends on PPOUT_j and PPOUT_l and vice-versa for the original graph in figure 1. Thus the information may flow from node j to node l via node k and vice-versa. However, after edge-splitting, IN_k ceases to depend on PPOUT_j (and PPOUT_l) since $\text{PPOUT}_{j'}$ does not depend on $\text{PPIN}_{j'}$. Effectively, the flow from nodes j and l to node k gets truncated at nodes j' and l' thereby reducing the complexity.

The formulations which try to achieve first, second, and the third contribution mentioned in section 1 by using the graph transformation of edge-splitting before performing analysis are : [6, 20, 22, 23, 26, 27, 35–39]. The other formulations [13, 19, 21, 47] insert new nodes by splitting the edges *on demand*, i.e. at the end of analysis after determining the need of placing computations along an edge. However, the effect is the same — flows get truncated at (some) critical edges which are split by inserting new nodes. Hence we will collectively refer to both the approaches as edge-splitting.

2.2.2 On the necessity and sufficiency of edge-splitting

The formulations based on edge-splitting claim to use edge-splitting for enhancing the effectiveness of PRE. However, they need edge-splitting for correctness also despite the fact that it is not a necessary condition for PRE. A careful examination reveals that these formulations derive their ability of performing bidirectional analysis from edge-splitting rather than their characterizations of flow dependencies.

We substantiate our claim as follows :

1. If edge-splitting is not performed, these formulations fail to produce correct result.
2. If edge-splitting is performed,

- Contribution 1 is very easily ensured without requiring an alternative formulation.
  PRE can be performed as a unidirectional data flow problem followed by a forward correction pass [13, 20, 22]. This can be explained using the Generalized Theory as follows :

  > Consider figure 1. The backward flow of information (i.e. flow beginning at the edge (k,l)) that would take place after forward flow (i.e. the flow along the edge (j,k)), gets truncated due to edge-splitting and the effect of forward flow is restricted only to the immediate successors in the *modified* graph. Thus the flows involved become "mostly backward" and data flow analysis could be performed as for a backward unidirectional data flow problem. The forward flow can be incorporated in one single traversal.

  It should be noted that this is achieved by using the *same data flow equations without requiring any alternative formulation.*
- Contribution 2 is automatically ensured by the virtue of edge-splitting without requiring an alternative formulation. This was suggested in the original formulation [44].
- Contribution 3 is easily ensured without requiring an alternative formulation.
  The desired notion of optimality which is a major claim in [26, 37, 47] can be achieved by performing an additional forward analysis. In other words, what is required is just an additional data flow analysis over and above (unidirectional) partial redundancy analysis rather than an *alternative* formulation. This approach is used in [21].

3. Though edge-spitting converts bidirectional flows of PRE into unidirectional flows, flows involved in some other bidirectional data flow problems may remain bidirectional in spite of edge-splitting — as an example, we present a variant of PRE in section 3. Such "truly" bidirectional data flow problems are not amenable to decomposition into a sequence of unidirectional data flow problems since :

   - the simplifying graph transformation of edge-splitting and
   - the favourable nature of data flows (viz. the absence of forward node flow in PRE),

   both together enable the conversion of bidirectional flows into unidirectional flows thereby obviating bidirectional analysis. In the situations where any of the two is absent, the approach of decomposition is infeasible.

We conclude that, in general, edge-splitting is neither necessary for performing bidirectional analysis nor sufficient for reducing the complexity of bidirectional analysis. For PRE, edge-splitting is not necessary but is sufficient for reducing the complexity and is an enabling transformation for enhancing profitability. Note that if edge-splitting is performed, gains in efficiency and profitability can be achieved within the same framework without requiring an alternative formulation.

3 A Truly Bidirectional Data Flow Problem

Code placement performed by partial redundancy elimination may be sub-optimal in some situations in that it may cause redundant hoisting of expressions, thus lengthening their live ranges and increasing register pressure. Redundant hoisting has been a matter of concern and several approaches have been proposed to avoid it [7, 13, 16, 19, 21, 26, 35, 37, 44, 47].

Redundant hoisting can be completely eliminated if no expression is ever hoisted to a node where it is partially redundant. This section modifies the PPOUT equation of partial redundancy elimination to incorporate this effect. Other equations (i.e. PPIN, INSERT, and REDUND) as also the semantics of all data flow properties remain same as in the original framework. The intuition behind this approach is quite simple : PRE *hoists* an expression from its original point of computation in order to convert a partial redundancy into a total redundancy. It is possible to *sink* the hoisted expression to those points where it is not partially redundant. Note

Local Data Flow Properties :

ANTLOC_i	Node i contains a computation of the expression, not preceded by a definition of any of its operands.
COMP_i	Node i contains a computation of the expression, not followed by a definition of any of its operands.
TRANSP_i	Node i does not contain a definition of any operand of the expression.

Global Data Flow Properties :

$\text{AVIN}_i/\text{AVOUT}_i$	The expression is available at the entry/exit of node i.
$\text{PAVIN}_i/\text{PAVOUT}_i$	The expression is partially available at the entry/exit of node i.
NOSINK_i	The expression cannot be sunk below node i.
$\text{PPIN}_i/\text{PPOUT}_i$	Computation of the expression may be placed at the entry/exit of node i.
INSERT_i	Computation of the expression should be inserted at the exit of node i.
REDUND_i	First computation of the expression existing in node i is redundant.

Data Flow Equations :

$$\text{AVIN}_i = \prod_{j \in pred(i)} (\text{AVOUT}_j) \qquad \text{AVOUT}_i = \text{COMP}_i + \text{TRANSP}_i \cdot \text{AVIN}_i$$

$$\text{PAVIN}_i = \sum_{j \in pred(i)} (\text{PAVOUT}_j) \qquad \text{PAVOUT}_i = \text{COMP}_i + \text{TRANSP}_i \cdot \text{PAVIN}_i$$

$$\text{PPIN}_i = \text{PAVIN}_i \cdot (\text{ANTLOC}_i + \text{TRANSP}_i \cdot \text{PPOUT}_i) \cdot \prod_{j \in pred(i)} (\text{AVOUT}_j + \text{PPOUT}_j) \qquad \text{PPOUT}_i = (\text{PPIN}_i + \text{NOSINK}_i) \cdot \prod_{k \in succ(i)} (\text{PPIN}_k)$$

$$\text{NOSINK}_i = \neg\text{PAVOUT}_i \cdot \Big(\prod_{k \in succ(i)} (\text{PAVIN}_k)\Big)$$

$$\text{INSERT}_i = \text{PPOUT}_i \cdot \neg\text{AVOUT}_i \cdot (\neg\text{PPIN}_i + \neg\text{TRANSP}_i) \qquad \text{REDUND}_i = \text{PPIN}_i \cdot \text{ANTLOC}_i$$

Figure 2: Data Flow Equations for Partial Redundancy Elimination without Redundant Hoisting.

that sinking should be considered for an expression along its hoisting path only. Thus an expression is never sunk below its original point of computation.[1]

Figure 2 provides the data flow equations. Sinking is achieved by incorporating the term $(\text{PPIN}_i + \text{NOSINK}_i)$ in the equation for PPOUT_i — data flow property NOSINK_i indicates the point where sinking should stop. It is easy to see how sinking occurs : When PPIN_i is **F**, i.e. *false*, the expression cannot be hoisted any further. The presence of PPIN_i in PPOUT_i equation tries to sink the expression (by setting PPOUT_i to **F**). Sinking stops if PPOUT_i is **T** due to NOSINK_i. Else, PPOUT_i becomes **F** setting the PPIN of its successors to **F** and the expression may be sunk further depending upon NOSINK property of the successors.

Figure 3 elaborates on this by comparing it with original partial redundancy elimination.

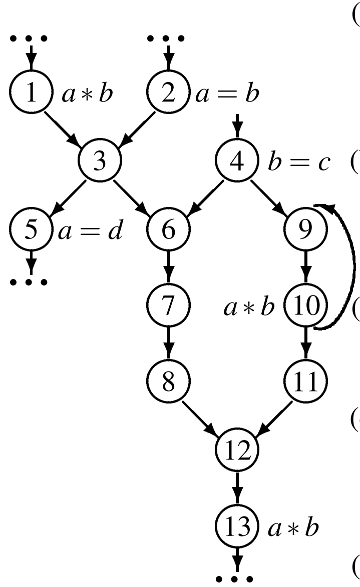
(a) *Original Partial Redundancy Elimination* [44].

Expression $a * b$ is not anticipated at the entry of node 5. Hence PPIN_5 = **F**. This causes flow of information [33] from node 5 to node 10 by setting $\text{PPOUT}_3 = \text{PPIN}_6 = \text{PPOUT}_4 = \text{PPIN}_9 = \text{PPOUT}_{10}$ = **F**. Since the expression cannot be hoisted into nodes 3 and 4, it is inserted in node 6 and 9. Thus, it is hoisted out of node 13 but cannot be hoisted out of the loop.

If the expression $a * b$ did not exist in node 10 then it would be inserted in nodes 6 and 11.

(b) *Original Partial Redundancy Elimination with Edge-splitting.*

[1]When it is sunk to the original point of computation, effectively, the expression is not hoisted at all.



(a) *Original Partial Redundancy Elimination (PRE)* [44].

Expression $a*b$ is hoisted from node 13 to node 6 but cannot be hoisted out of the loop (gets hoisted from node 10 to node 9).

(b) *Original PRE with Edge-splitting.*

New nodes are inserted along edges (3,6), (4,6), and (4,9). Expression $a*b$ is hoisted from nodes 13 and 10 to these new nodes.

(c) *PRE with Edge-placement* [13].

Expression $a*b$ will be placed in node 8 and edge (4,9).

(d) *PRE without Redundant Hoisting.*

The expression is not hoisted since there is no program point where the hoisted expression would be safe but would not be partially redundant.

(e) *PRE without Redundant Hoisting in presence of Edge-splitting.*

The expression gets hoisted out of the loop but not out of node 13.

Figure 3: An Example of Partial Redundancy Elimination without Redundant Hoisting

New nodes are inserted along edges (3,6), (4,6), and (4,9). Expression $a*b$ is hoisted from nodes 13 and 10 to these new nodes. Note that the flow of information from node 5 gets truncated at the new node along edge (3,6) — even if the PPIN of the new node is **F**, its PPOUT cannot become **F**.

(c) *Partial Redundancy Elimination with Edge-Placement* [13].

Expression $a*b$ is inserted in node 8 and along the edge (4,9). This is because it avoids redundant hoisting along the path $(6,7,8)$.

(d) *Partial Redundancy Elimination without Redundant Hoisting.*

Apart from the **F** properties in step (a) above, PPIN and PPOUT for nodes 6, 7, 8, 11, 12, and 13 are **F**. NOSINK_4 = **T** but PPOUT_4 = **F** due to PPIN_6. The expression is not hoisted at all since there is no program point where the hoisted expression would be safe but would not be partially redundant.

If the expression did not exist in node 10 then NOSINK_{11} would be **T**. However, NOSINK_8 would be **F** due to PAVOUT_8 which is **T**. This would cause PPOUT_8, PPIN_{12}, PPOUT_{12}, and hence PPIN_{13} to be **F** prohibiting hoisting.

(e) *Partial Redundancy Elimination without Redundant Hoisting in presence of Edge-splitting.*

PPOUT of the new node along edge (4,9) remains **T** due to NOSINK. Hence PPIN and PPOUT of nodes 9 and 10 remain **T**. The expression gets hoisted out of the loop but not out of node 13.

Note that unlike in part (b) above, the information flow *does not* truncate at the newly created nodes since their PPOUT properties depend on their PPIN properties. The flow stops at the new node along edge (4,9) since NOSINK property of this node is **T**.

Salient features of this variant of partial redundancy elimination are as follows :

- The semantics of all data flow properties remain exactly the same as in the original framework.
- The data flow equations remain same except for the addition of one term in the PPOUT equation.

- The computation of NOSINK does not require data flow analysis; it only requires one pass.

- The resulting data flow equations are "truly" bidirectional in nature and as such are not amenable to the adhoc approache of edge-splitting for reducing the complexity.[2] However, they can be explained using the Generalized Theory [32, 33].

- It must be noted, however, that since this variant has been aimed at exposition rather than serving a practical need, it misses out some potentially profitable optimizations. For instance, if the expression $a * b$ is hoisted to some node on the path (6,7,8), partial redundancy along the path (9,10,11,12,13) is eliminated. However, this does not happen since this variant puts stringent conditions on hoisting.

4 Conclusions and Future Research Directions

Partial redundancy elimination and similar analyses have been performed in many innovative ways. However, most of the approaches are adhoc in that they cannot go beyond this class of data flow problems. It is necessary to look at bidirectional data flow problems at a general level independently of the partial redundancy elimination framework since simultaneous bidirectional dependencies constitute a powerful mechanism for capturing complex data flow semantics. Research efforts will be more productive if they are directed at harnessing this power rather than avoiding it. Practical techniques for performing bidirectional analysis and rigourous theoretical foundations for explaining the process have been provided in [18, 32, 33, 55].

We suggest some directions for research in bidirectional data flow analysis :

- The PRE variant presented in this paper is not very practical since it misses out some optimization opportunities. However, there may be some other application where this kind of strict[3] data flow semantics is desirable. We invite researchers to find out newer applications of bidirectional data flows.

- The correlation between the predicted bound for bidirectional flows and practical number of iteration seems to be quite good for original partial redundancy elimination [18]. However, in the case of truly bidirectional data flows, it does not seem to be so good. This needs to be investigated.

- Another interesting direction of exploration is the elimination methods of data flow analysis for bidirectional flows using the concept of cyclic information flow graphs [55].

References

[1] G. Agrawal. Interprocedural partial redundancy elimination with application to distributed memory compilation. *IEEE Transactions on Parallel and Distributed Systems*, 9(7):609–625, 1998.

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

[3] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of ACM*, 19(3):137–147, 1977.

[4] R. Bodik and R. Gupta. Partial dead code elimination using slicing transformations. In *Proceedings of ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 159–170, 1997. Also Published as *SIGPLAN Notices*, 32(5).

[2] We invite the researchers to try to perform analysis for the proposed variant by decomposing it into a sequence of unidirectional data flows.

[3] This variant puts more constraints on PPOUT property.

[5] R. Bodik, R. Gupta, and M. L. Soffa. Complete removal of redundant computations. In *Proceedings of ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 1–14, 1998. Also Published as *SIGPLAN Notices*, 33(5).

[6] P. Briggs and K. D. Kooper. Effective partial redundancy elimination. In *Proceedings of ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 159–170, 1994. Also Published as *SIGPLAN Notices*, 29(6).

[7] F. C. Chow. *A portable machine-independent global optimizer — Design and measurements*. PhD thesis, Computer Systems Laboratory, Stanford University, 1983.

[8] F. C. Chow. Minimizing register usage penalty at procedure calls. In *Proceedings of SIGPLAN '88 Symposium. on Compiler Construction*, pages 85–94, 1988. Also Published as *SIGPLAN Notices*, 23(7).

[9] F. C. Chow, S. Chan, R. Kennedy, S. M. Liu, R. Lo, and P. Tu. A new algorithm for partial redundancy elimination based on SSA form. In *Proceesings of ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 273–286, 1997. Also Published as *SIGPLAN Notices*, 32(5).

[10] P. Cousot. Semantic foundations of program analysis, 1981. In [46].

[11] P. Cousot and R. Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4^{th} Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[12] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6^{th} Annual ACM Symposium on Principles of Programming Languages*, pages 269–282, 1979.

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

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

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

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

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

[18] D. M. Dhamdhere and Uday P. Khedker. Complexity of bidirectional data flow analysis. In *Proceedings of the 20^{th} Annual ACM Symposium on Principles of Programming Languages*, pages 397–408, 1993.

[19] D. M. Dhamdhere and Harish Patil. An elimination algorithm for bidirectional data flow analysis using edge placement technique. *ACM Transactions on Programming Languages and Systems*, 15(2):312–336, 1993.

[20] D. M. Dhamdhere, B. K. Rosen, and F. K. Zadeck. How to analyze large programs efficiently and informatively. In *Proceedings of ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 212–223, 1992. Also Published as *SIGPLAN Notices*, 27(7).

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

[22] K. Drechsler and M. P. Stadel. A solution to a problem with Morel and Renvoise's "Global optimizations by suppression of partial redundancies". *ACM Transactions on Programming Languages and Systems*, 10(4):635–640, 1988.

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

[24] S. Graham and M. Wegman. A fast and usually linear algorithm for global data flow analysis. *Journal of ACM*, 23(1):172–202, 1976.

[25] M. Gupta, E. Schonberg, and H. Srinivasan. A unified framework for optimizing communication in data-parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 7(7):689–704, 1996.

[26] Max Hailperin. Cost-optimal code motion. *ACM Transactions on Programming Languages and Systems*, 20(6):1297–1322, 1998.

[27] R. Hanxleden and K. Kennedy. GIVE-N-TAKE — A balanced code placement framework. In *Proceedings of ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 107–120, 1994. Also Published as *SIGPLAN Notices*, 29(6).

[28] M. S. Hecht. *Flow Analysis of Computer Programs*. Elsevier North-Holland Inc., 1977.

[29] S. M. Joshi and D. M. Dhamdhere. A composite algorithm for strength reduction and code movement : Part I. *International Journal of Computer Mathematics*, 11(1):21–44, 1982.

[30] S. M. Joshi and D. M. Dhamdhere. A composite algorithm for strength reduction and code movement : Part II. *International Journal of Computer Mathematics*, 11(2):111–126, 1982.

[31] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–318, 1977.

[32] Uday P. Khedker. *A Generalized Theory of Bit Vector Data Flow Analysis*. PhD thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Bombay, 1997.

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

[34] G. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM Symposium on Principles of Programming Languages*, pages 194–206, 1973.

[35] J. Knoop, O. Ruthing, and B. Steffen. Lazy code motion. In *Proceedings of ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 224–234, 1992. Also Published as *SIGPLAN Notices*, 27(7).

[36] J. Knoop, O. Ruthing, and B. Steffen. Lazy strength reduction. *Journal of Programming Languages*, 1(1):71–91, 1993.

[37] J. Knoop, O. Ruthing, and B. Steffen. Optimal code motion. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, 1994.

[38] J. Knoop, O. Ruthing, and B. Steffen. Partial dead code elimination. In *Proceedings of ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 147–158, 1994. Also Published as *SIGPLAN Notices*, 29(6).

[39] J. Knoop, O. Ruthing, and B. Steffen. The power of assignment motion. In *Proceedings of ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 233–245, 1995. Also Published as *SIGPLAN Notices*, 30(6).

[40] P. Kolte and M. Wolfe. Elimination of redundant array subscript checks. In *Proceedings of ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 270–278, 1995. Also Published as *SIGPLAN Notices*, 30(6).

[41] R. Lo, F. C. Chow, R. Kennedy, S. M. Liu, and P. Tu. Register promotion by sparse partial redundancy elimination of loads and stores. In *Proceesings of ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 26–37, 1997. Also Published as *SIGPLAN Notices*, 33(5).

[42] T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks. *Acta Informatica*, 28:121–163, 1990.

[43] S. P. Masticola, T. J. Marlowe, and B. G. Ryder. Lattice frameworks for multi-source and bidirectional data flow problems. *ACM Transactions on Programming Languages and Systems*, 17(5):777–803, 1995.

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

[45] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishing Co., 1997.

[46] S. S. Muchnick and N. D. Jones. *Program Flow Analysis : Theory and Applications*. Prentice-Hall Inc., 1981.

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

[48] B. K. Rosen. Monoids for rapid data flow analysis. *SIAM Journal of Computing*, 9(1):159–196, 1980.

[49] B. K. Rosen, M. N. Wegmen, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 12–27, 1988.

[50] B. G. Ryder and M. C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18:277–316, 1986.

[51] David A. Schmidt. Data flow analysis is model checking of abstract interpretation. In *Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages*, pages 38–48, 1998.

[52] V. C. Shreedhar, G. R. Gao, and Y-F Lee. A new framework for exhaustive and incremental data flow analysis using D-J graphs. In *Proceedings of ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 278–290, 1996. Also Published as *SIGPLAN Notices*, 31(6).

[53] L. T. Simpson. *Value-Driven Redundancy Elimination*. PhD thesis, Department of Computer Science, Rice University, Houston, 1996.

[54] A. Sorkin. Some comments on a solution to a problem with Morel and Renvoise's "Global optimizations by suppression of partial redundancies". *ACM Transactions on Programming Languages and Systems*, 11(4):666–668, 1989.

[55] Ashok Sreeniwas. *Program Transformations and Representations*. PhD thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Bombay, 1998.

[56] R. E. Tarjan. Fast algorithms for solving path problems. *Journal of ACM*, 28(3):594–614, 1981.