Interprocedural Partial Redundancy Elimination with Application to Distributed Memory Compilation

Gagan Agrawal
Department of Computer and Information Sciences
University of Delaware
Newark DE 19716
(302)-831-2783
{agrawal@cis.udel.edu}

Abstract
Partial Redundancy Elimination (PRE) is a general scheme for suppressing partial redundancies which encompasses traditional optimizations like loop invariant code motion and redundant code elimination. In this paper we address the problem of performing this optimization interprocedurally. We present an Interprocedural Partial Redundancy Elimination (IPRE) scheme based upon a new concise full program representation. Our method is applicable to arbitrary recursive programs.

We use interprocedural partial redundancy elimination for placement of communication and communication preprocessing statements while compiling for distributed memory parallel machines. We have implemented our scheme as an extension to the Fortran D compilation system. We present experimental results from two codes compiled using our system to demonstrate the useful of IPRE in distributed memory compilation.

Keywords: Distributed Memory Compilation, Partial Redundancy Elimination, Interprocedural Analysis, Communication Optimizations, Irregular Applications
1 Introduction

Partial Redundancy Elimination (PRE) is a well known technique for optimizing code by suppressing partially redundant computations. It encompasses traditional optimizations like loop invariant code motion and redundant computation elimination. It is widely used in optimizing compilers for performing common subexpression elimination and strength reduction. More recently, it has been used for more complex code placement tasks like placement of communication statements while compiling for parallel machines [18, 23].

A number of schemes for partial redundancy elimination have been proposed in literature [15, 16, 32, 30, 37], but are largely restricted to optimizing code within a single procedure. All these schemes perform data flow analysis on Control Flow Graph (CFG) of the procedure. In this paper, we address the problem of performing partial redundancy elimination interprocedurally.

Our study of Interprocedural Partial Redundancy Elimination (IPRE) was initially motivated by the problem of placement of communication and communication preprocessing statements in distributed memory compilation. Making communication efficient is a major challenge for the compilers for distributed memory machines. It is often not possible to determine the set of data elements to be communicated at compile time. This may happen because data distribution may not be known at compile time, exact loop bounds and strides may not be compile time constants or the data may be accessed through indirection arrays whose contents are not known at compile time. In these cases, it may not be possible for the compiler to ensure optimized communication. The compiler can, however, insert calls to runtime routines which can determine the data to be communicated and obtain optimized communication performance. In this approach, a communication preprocessing routine is called at runtime which computes the set of data elements to be communicated. Collective Communication routines can use the result of runtime preprocessing to communicate between the processors.

Compiler techniques have been developed for recognizing data access patterns at compile time and inserting calls to appropriate routines [1, 22]. Runtime preprocessing can be expensive, so it is important to reuse the result of preprocessing whenever possible. For example, if the set of data elements to be accessed does not change over multiple iterations of a sequential loop, then the runtime preprocessing needs to be done only once. Similarly, if the same data elements need to be accessed by several different loops in the program, then the result of runtime preprocessing can be used across these loops. Partial redundancy elimination can be applied interprocedurally to solve these optimization problems.

In this paper, we present an interprocedural partial redundancy elimination algorithm. Our method is applicable to arbitrary recursive programs whose procedures have arbitrary control flow. We have developed a new full program representation for performing data flow analysis for partial redundancy elimination. This representation allows for efficient data flow analysis, while maintaining sufficient precision to ensure correctness and safety of transformations and allowing useful transformations.

The rest of this paper is organized as follows. In Section 2, we discuss how interprocedural partial redundancy elimination is required for placement of communication statements and communication preprocessing statements while compiling for distributed memory parallel machines. In Section 3, we briefly review an intraprocedural partial redundancy elimination scheme. The program representation used in interprocedural data flow analysis is stated in Section 4. Interprocedural partial redundancy elimination is presented in Section 5. In Section 6, we present an overview of the correctness proof of our method\(^1\).

\(^1\)Note to the Reviewers: The detailed proofs have not been included in this version of the paper to meet the page limit.
Experimental results on usefulness of the method in distributed memory compilation are presented in Section 7. In Section 8, we mention related work. We conclude in Section 9.

2 Distributed Memory Compilation

In compiling programs for execution on distributed memory parallel machines, an important consideration is optimizing communication between processors. Since several existing machines have relatively large communication latencies, communication overhead can be reduced if message aggregation is done, i.e. each processor sends a small number of large messages. There are several scenarios in which the set of data elements to be communicated between the processors cannot be determined at compile-time:

- **Irregular Codes**: Data may be accessed using indirection arrays, i.e. contents of an array A may be accessed depending upon contents of an array IA in a data parallel loop. We refer to the codes in which the data is accessed using indirection arrays as **irregular codes**.

- **Unknown Data Distribution**: Data distribution of the arrays accessed or modified in a parallel loop may not be known to the compiler. Data parallel languages like High Performance Fortran (HPF) [31] allow *Redistribute* and *Realign* directives for modifying distribution of arrays during execution of the program. In presence of such directives, the compiler may not always have sufficient information to determine the communication required for a given parallel loop.

- **Unknown Number of Processors**: Number of processors on which the program is to be executed may not be known till runtime. This can happen when a number of data parallel tasks are executed in parallel and the number of processors available for each task may not be statically known [3, 38], or when a data parallel program is executed on a non-dedicated network of workstations and the number of workstations available may not be known in advance [17].

- **Complex Symbolic Loop Bounds and Strides**: A data parallel loop may involve complex symbolic loop bounds and strides and it may not be possible to analyze the set of data elements to be communicated between the processors at compile time.

In all the above cases, message aggregation can be done by using communication preprocessing. We discuss this in detail, separately in the context of irregular codes (i.e. Case I above) and other codes (i.e. Cases II-IV above).

2.1 Compiling Irregular Codes

In Figure 1, we show a parallel loop in which array Y is accessed through an indirection array IA. The elements of the array Y that need to be communicated depends upon the contents of array IA, which is not known at compile-time.

Runtime preprocessing can be used to optimize communication. CHAOS/PARTI routines for irregular problems [13] provide such functionality. In Figure 1, we also show how SPMD code can be generated for the given irregular loop. A communication schedule is generated by a call to *Irreg_Sched*, which analyzes the contents of array IA to determine the exact communication required. The required data elements are sent or received by the *Data_Move* routine.

A technical report with the detailed proofs is available.
Real $X(n)$, $Y(m)$ ! data arrays
Integer $IA(n)$ ! indirection array

C Original HPF code
For all $i = 1, n$
\[
X(i) = X(i) + Y(IA(i))
\]

C Compiled Code
C Build the required schedule
Sched = Irreg. Sched(., parameters..)
C Communicate data using the schedule built above
Call Data. Move($Y$, Sched)

do 10 i = 1, n
\[
X(i) = X(i) + Y(IA_{local}(i))
\]
10 continue

Figure 1: Compiling a HPF irregular loop

C Original HPF code
For all $(i = a:b; 2, j = c:d)$
\[
A(i,j) = B(n^{*}j,i) + C(i,j)
\]

C Compiled code
C
Sched = Reg. Sched(., parameters..)
Call Data. Move($B$, Sched, Temp)
C Compute loop bounds ($L1,H1,L2,H2$)

do 10 i = $L1$, $H1$
\[
\]
do 10 j = $L2$, $H2$
\[
A(i,j) = Temp(i,j) + C(i,j)
\]
10 continue

Figure 2: Compiling HPF loop when loop bounds are not compile time constants

2.2 Compiling Regular Codes

In a parallel loop, if data is accessed without using indirection arrays, all loop bounds and strides are compile time constants and data distribution is known at compile time; then the compiler can determine new loop bounds on each processor and the exact set of data elements to be communicated. In this case, the compiler can perform various communication optimizations [25, 26]. However, if all information is not available at compile time, then the compiler may need to insert symbolic calculations in the generated code to determine new loop bounds and data sets to be communicated. This can become very complicated, especially when the data is not block distributed, data distribution is not known or there are (possibly) non-unit strides. The compilation task can be simplified by inserting calls to runtime routines which can determine new loop bounds, exact data sets to be communicated and can perform communication optimizations. In practice, application codes are written to execute on varying problem sizes or problem instances and therefore loop bounds and strides may not be compile time constants, or data distribution may not be known at compile time.

In Figure 2, we show how a data parallel loop can be compiled when the loop bounds are not compile time constants. In this example, the values of $a, b, c, d$ and $n$ are not known at compile time. Let us assume that the arrays $A$, $B$ and $C$ have identical sizes and distribution. A schedule building routine $Reg. Sched$, available in the Multiblock PARTI Library we have developed [39], can be used for generating the communication schedule. The communication schedule $Sched$ is used by $Data. Move$. Since the loop bounds of the sequential program are not known at compile time, the new loop bounds for the SPMD code need to be determined during execution at each processor. Even if the loop bounds are known at compile time, runtime preprocessing may be required if complete information about the distribution of
arrays A, B and C is not available to the compiler.

The same problem arises when the number of processors on which a data parallel loop is to be executed is not known at compile-time. This can happen mainly for two reasons: combining task and data parallelism and use of network of workstations. It has been widely recognized that exploiting both data parallelism and task parallelism is important for obtaining good performance on several classes of applications [3, 38]. If both data and task parallelism is to be used, then each data parallel task is executed on a subset of the available processor set. The number of processors available on each data parallel task may not be known at compile-time. Another increasing trend is to use a network of workstations for executing parallel programs. Number of workstations available for a parallel job may depend upon the existing load at the time of execution and may not be statically known at compile-time.

2.3 Optimization Problems

In both regular and irregular applications, runtime preprocessing can be used to optimize communication whenever the data set to be communicated cannot be determined at compile time. Runtime preprocessing (i.e. calls to routines Reg_Sched or Irreg_Sched) can be expensive. Large application programs often present opportunities for reusing schedule several times [13, 14]. Large scientific and engineering computations are often iterative in nature, which means that there is a main time step loop to iterate over a set of routines performing the computation. It is often possible to use the same set of schedules over different iterations. Also, many computational loops, possibly spread over several subroutines, may require the same set of data elements to be communicated. In this case, same schedule can be used again and the overhead of runtime preprocessing can be reduced. In some cases, the data communicated may not have been modified, so a loop executed later may just use the data communicated for executing an earlier loop.

We can, therefore, formulate two different optimization problems for distributed memory compilation. Communication Preprocessing Placement means optimized placement of communication preprocessing routines and their reuse to reduce the overhead. Communication Placement means optimized placement of collective communication routines and avoiding redundant communication.

Partial redundancy elimination can be applied interprocedurally for solving both these optimization problems. Partial redundancy elimination encompasses loop invariant code motion and redundant code elimination and has been widely used interprocedurally to improve the runtime performance of codes. We believe that it can be applied interprocedurally to optimize placement of communication preprocessing statements and communication statements.

3 Intraprocedural Redundancy Elimination

In this section, we review an intraprocedural partial redundancy elimination scheme. To the extent possible, we use the same terminology in describing our interprocedural scheme as commonly used for intraprocedural partial redundancy elimination schemes. It is, therefore, useful to revisit the details of an intraprocedural scheme.

The first partial redundancy elimination scheme was presented by Morel and Renvoise [32]. This scheme has been further extended and refined by Dhamdhere [15], Drechsler [16], Knoop et. al. [30] and Sorkin [37]. The details of interprocedural redundancy elimination we present are derived from the intraprocedural node based method of Dhamdhere [15], also referred to as Modified Morel Renvoise
\begin{align*}
\text{AVIN}(i) &= \begin{cases} 
\text{false} & \text{if } i \text{ is entry basic block} \\
\prod_{p \in \text{pred}(i)} \text{AVOUT}(p) & \text{otherwise}
\end{cases} \\
\text{AVOUT}(i) &= \text{COMP}(i) + \text{TRANS}(i) \cdot \text{AVIN}(i) \\
\text{PAVIN}(i) &= \begin{cases} 
\text{false} & \text{if } i \text{ is entry basic block} \\
\sum_{p \in \text{pred}(i)} \text{PAVOUT}(p) & \text{otherwise}
\end{cases} \\
\text{PAVOUT}(i) &= \text{COMP}(i) + \text{TRANS}(i) \cdot \text{PAVIN}(i) \\
\text{PPIN}(i) &= \text{PAVIN}(i) \cdot (\text{ANTLOC}(i) + \text{TRANS}(i) \cdot \text{PPOUT}(i)) \\
&\quad \times \prod_{p \in \text{pred}(i)} (\text{PPOUT}(p) + \text{AVOUT}(p)) \\
\text{PPOUT}(i) &= \begin{cases} 
\text{false} & \text{if } i \text{ is exit basic block} \\
\prod_{s \in \text{succ}(i)} \text{PPIN}(s) & \text{otherwise}
\end{cases} \\
\text{INSERT}(i) &= \text{PPOUT}(i) \cdot \text{AVOUT}(i) \cdot (\neg \text{PPIN}(i) + \neg \text{TRANS}(i)) \\
\text{DEL}(i) &= \text{ANTLOC}(i) \cdot \text{PPIN}(i)
\end{align*}

Figure 3: MMRA scheme for Intraprocedural Partial Redundancy Elimination

Algorithm (MMRA). The data flow equations used in the scheme are presented in Figure 3. The terms used in the data flow equations are explained below.

Local data flow properties:

\text{ANTLOC}(i): \text{Node } i \text{ contains an occurrence of computation } C \text{ not preceded by a definition of any of its operands.}

\text{COMP}(i): \text{Node } i \text{ contains an occurrence of computation } C \text{ not followed by a definition of any of its operands.}

\text{TRANS}(i): \text{Node } i \text{ does not contain a definition of any operand of computation } C.

Global data flow properties:

\text{AVIN}(i)/\text{AVOUT}(i): \text{Computation } C \text{ is available at the entry/exit of node } i.

\text{PAVIN}(i)/\text{PAVOUT}(i): \text{Computation } C \text{ is partially available at the entry/exit of node } i.

\text{PPIN}(i)/\text{PPOUT}(i): \text{Computation of } C \text{ may be placed at entry/exit of node } i.

\text{DEL}(i): \text{Occurrence of } C \text{ in node } i \text{ is redundant}

\text{INSERT}(i): \text{A computation of } C \text{ should be placed at the exit of node } i.

All terms used are for a particular computation, e.g \text{AVIN}_C(i) \text{ is the availability of the computation } C \text{ at the beginning of node } i. \text{Whenever there is no scope for ambiguity, the subscript is dropped (as in the equations given in this paper).}
PRE considers subexpressions or computations as candidates for placement. Transparency of a basic block means that the none of variables involved in the computation are modified in the basic block. Based upon transparency, two properties, availability and partial availability are computed for beginning and end of each basic block (denoted respectively as AVIN, PAVIN, AVOUT and PAVOUT for each basic block). Availability of a computation at a point \( P \) in a procedure means that this computation is currently placed at all the paths leading to \( P \) and if this computation is placed at \( P \), it will have the same result as the last computation on any of the paths. Partial availability is a weaker property, which means that the computation is currently placed on at least one control flow path leading to \( P \) and if it is placed at \( P \), it will have the same result as the last computation on at least one of the paths. A computation placed at \( P \) is partially redundant if it is partially available at \( P \).

A computation is available at the entry of a basic block if it is available at the exit of all the predecessor basic blocks (see equation 3.1). A computation is available at the end of a basic block if it is available at the beginning of the basic block and none of the operands are modified in the basic block, or, alternatively, there is an occurrence of this computation in this basic block, not followed by any definition of the operands (equation 3.2). A computation is partially available at the entry of a basic block if it is partially available at the exit of at least one predecessor block (equation 3.3).

After computing availability and partial availability, properties PPIN (possible placement at the beginning) and PPOUT (possible placement at the end) are computed for each basic block in the program. PPIN reflects the fact that it is feasible and profitable to hoist the computation occurring in this node (or a computation which has been hoisted into this node). PPOUT indicates that it is safe to place the computation at the exit of this node. The equations for placement can be explained as follows. In equation 3.6, the \( \prod \) term ensures safety in placing an expression at the exit of the node. The \( \prod \) term in equation 3.5 ensures availability of the expression at the entry of this node in the optimized program. The term PAVIN determines the profitability of hoisting a computation out of this node. This term avoids redundant code hoisting for almost all cases for any real program; however, it does not guarantee that redundant code motion will not occur. In the original MMRA scheme [15], an additional term is used to further prevent redundant code hoisting. This term still does not guarantee that no redundant code hoisting occurs. For simplicity, we do not include this additional term in our presentation.

INSERT determines if a computation is to be inserted at the end of a block as a result of the optimization and DEL determines if the computation in this node has become redundant and can be deleted.

All PRE schemes satisfy correctness and safety. Correctness means that no computation is deleted unless it is redundant or has been made redundant by insertion of a new computation. Safety means that insertion of a new computation does not increase the number of occurrences of the computation along any path in the CFG. This implies that if any new computation is placed at any node, then a computation is deleted along all paths in the CFG starting from this node before any further occurrences of the computation.

Calculating availability requires unidirectional data flow analysis, since the data flow property for a particular node only depends upon the data flow property of its predecessor nodes. For the procedures whose control flow graph is reducible, such data flow properties can be efficiently solved using elimination methods (Interval analysis is one such elimination method [7]). Such methods take \( O(E) \) bit-vector steps, where \( E \) is the number of edges in the control flow graph. For any arbitrary shaped control flow graph, the worst case time required for computation of unidirectional data flow property is \( O(N^2) \), where \( N \) is
the number of nodes in the graph. In practice, unidirectional data flow problems may converge much more quickly.

Calculating placement properties (PPIN and PPOUT) requires bidirectional analysis, in the sense that data flow property for a particular node may depend upon both the predecessors and successors of this node. These data flow properties cannot be solved efficiently using elimination methods. The worst case time required for solving these equations iteratively is again \( O(N^2) \), where the number of nodes in the graph is \( N \).

4 Program Representation

In this section, we introduce the concise full program representation we use for data flow analysis. A number of different program representations have been used for various interprocedural data flow problems. The most commonly used program representation is a Call Graph [19]. A call graph is a directed multi graph, which has a single node for each procedure in the program. A directed edge from node \( i \) to node \( j \) means that procedure \( i \) calls procedure \( j \). No information is available in a call graph about flow of control between different call sites within a single procedure. Call Graph is used for solving flow-insensitive interprocedural problems. Flow-insensitive analysis calculates data-flow effects at call sites without considering the control flow paths that will be taken through individual procedures. In contrast, flow-sensitive analysis derives the effects common to each distinct control flow path that reaches a location. Clearly, call graph does not include sufficient information to allow flow-sensitive data flow analysis.

Myer's SuperGraph [34] was the first program representation used for flow-sensitive interprocedural analysis. SuperGraph is constructed by linking control flow graphs of subroutines by inserting edges from call site in the caller to start node in callee. The total number of nodes in SuperGraph can get very large and consequently the solution may take much longer time to converge.

We believe that it is important to consider concise representations, even if they result in loss of accuracy, to a certain level. Most of the existing commercial compilation systems do not consider interprocedural analysis currently. As interprocedural analysis may be incorporated in future commercial systems, we believe that the focus will be on techniques which allow most of the useful optimizations at relatively low compilation overhead, rather than performing the most accurate analysis at high overheads. With this practical motivation, we have developed a new concise full program representation which is more detailed than a call graph.

4.1 Definition

We assume that a variable is either global to the entire program or is local to a single procedure. We further assume that all parameters are passed by reference. We do not consider the possibility of aliasing in our discussion.

We assume, for convenience, that a procedure can only be called through a separate call statement. Each procedure has one or more return statements, which end the call of this procedure. We define a basic block to consist of consecutive statements in the program text without any procedure calls or return statements, and no branching except at the beginning and end. A procedure can then be partitioned into a set of basic blocks, a set of call statements and a set of return statements. Each call statement is a
call site of the procedure called there. In general, a procedure can be called at several call sites in the program.

In our program representation, the main idea is to construct blocks of code within each procedure. A block of code comprises of basic blocks which do not have any call statement between them. In the directed graph we define below, each edge $e$ corresponds to a block of code $B(e)$. The nodes of the graph help clarify the control flow relationships between the blocks of code.

**Full Program Representation**: ($FPR$) is a directed multigraph $G = (V, E)$, where the set of nodes $V$ consists of an entry node and a return node for each procedure in the program. For procedure $i$, the entry node is denoted by $s_i$ and the return node is denoted by $r_i$. Edges are inserted in the following cases:

1. Procedures $i$ and $j$ are called by procedure $k$ at call sites $cs_1$ and $cs_2$ respectively and there is a path in the CFG of $k$ from $cs_1$ to $cs_2$ which does not include any other call statements. Edge $(r_i, s_j)$ exists in this case. This edge is said to be associated with call site $cs_1$ at its start and with call site $cs_2$ at its end. The block of code $B(e)$ consists of basic blocks of procedure $k$ which may be visited in any control flow path $fp$ from $cs_1$ to $cs_2$, such that the path $fp$ does not include any other call statements.
Program Foo
\[ a = 1 \]
\[ \textbf{Do} \ i = 1, 100 \]
\[ \quad \text{Call } P(a, b) \ldots \text{cs1} \]
\[ \quad \text{Call } Q(c) \ldots \text{cs2} \]
\[ \text{Enddo} \]
\[ \text{Call } Q(c) \ldots \text{cs3} \]
\[ \text{Call } P(a, c) \ldots \text{cs4} \]
\[ \textbf{if } \text{cond} \text{ then} \]
\[ \quad \text{Call } Q(a) \ldots \text{cs5} \]
\[ \textbf{Endif} \]
\[ \text{Call } R(a, c) \ldots \text{cs6} \]
\[ \textbf{End} \]

Procedure \( P(x, y) \)
\[ \text{Sched}(x, y) \]
\[ \ldots \text{other computations} \ldots \]
\[ \textbf{End} \]

Procedure \( Q(z) \)
\[ z = \ldots z \ldots \]
\[ \textbf{End} \]

Procedure \( R(y, z) \)
\[ \text{Sched}(y, z) \]
\[ \ldots \text{other computations} \ldots \]
\[ \textbf{End} \]

Figure 4: An Example Program. A call site number is marked for each call site.

Figure 5: \( FPR \) for Program in Left. Edge numbers and call sites at which edges start/end (whenever applicable) are marked in the Figure.
2. Procedure $i$ calls procedure $j$ at call site $cs$ and there is a path in the CFG of $i$ from the entry node of procedure $i$ to $cs$ which does not include any other call statements. In this case, edge $(s_i, s_j)$ exists. This edge is said to be associated with call site $cs$ at its end. The block of code $B(e)$ consists of the basic blocks of procedure $i$ which may be visited in any control flow path $fp$ from start of $i$ to $cs$, such that the path $fp$ does not include any other call statement.

3. Procedure $j$ calls procedure $i$ at call site $cs$ and there is a path in the CFG of $j$ from call site $cs$ to a return statement within procedure $j$ which does not include any other call statements. In this case, edge $(r_i, r_j)$ exists. This edge is said to be associated with call site $cs$ at its start. The block of code $B(e)$ consists of the basic blocks of procedure $j$ which may be visited in any control flow path $fp$ from $cs$ to a return statement of $j$, such that the path $fp$ does not include any call statements.

4. In a procedure $i$, there is a possible flow of control from entry node to a return statement, without any call statements. In this case, edge $(s_i, r_i)$ exists. The block of code $B(e)$ consists of the basic blocks of procedure $i$ which may be visited in any control flow path $fp$ from entry of $i$ to a return statement in $i$, such that the path $fp$ does not include any call statements.

An example program and its FPR are shown in Figures 4 and 5 respectively. In Figure 5, the blocks of codes $B(4)$, $B(9)$ and $B(11)$ comprise of all the basic blocks in procedures $P$, $Q$ and $R$ respectively. Block of code corresponding to all other edges comprise of the basic blocks from the main procedure. e.g. $B(1)$ comprises of statement “$a = 1$” and the loop header, $B(2)$ comprises of the end of the do loop and the loop header.

A block of code is a unit of placement in our analysis, i.e. we initially consider placement only at the beginning and end of a block of code\(^2\). Note that a basic block in a block of code may or may not be visited along a given control flow path from source to sink of the edge, and similarly, a basic block may belong to several blocks of code. This is taken into account during intraprocedural analysis done for determining final local placement, which we discuss in Section 5.5.

### 4.2 Candidates for Placement

We consider only the placement of pure functions. A pure function takes a number of parameters as input and produces a single result or output, without any side-effects or change in the value of inputs, or reading any variables except its parameters. Note that our definition of a pure function is different from the definition used in HPF [31], where accesses to global variables are allowed. In general, any subexpression can also be viewed as a pure function. In practice, one may want to focus on placement of only certain high cost functions, like communication statements and communication preprocessing statements in the case of distributed memory compilation.

A particular invocation of a pure function is considered for hoisting out of the procedure only if none of the parameters of the pure function is modified along any path from the start of the procedure to this invocation of the pure function and the invocation of pure function is not enclosed by any conditional or loop. In other words, we can consider the call to these pure functions to be placed at the beginning of the procedure, before any other computations. A particular invocation of a pure function is referred to

\(^2\)This is different from intraprocedural PRE in which placement is considered at beginning and end of node (basic block) of the graph.
as candidate if it is considered for interprocedural placement. We refer to the list of parameters of this pure function as the list of influencers of the candidate.

For presenting our scheme, we introduce the notion of procedure invocation.

Procedure Invocation. Invocation of a procedure \( p \) is a set of procedures \( S(p) \) such that

(a) The procedure \( p \) is in the set \( S(p) \).

(b) Any procedure which is called by a procedure in the set \( S(p) \) is in \( S(p) \).

When we refer to a procedure invocation \( p \), we mean the set of procedures in the set \( S(p) \) as defined above.

The following information is pre-computed and assumed to be available during our analysis phase. For each edge in \( FPR \), we compute all the variables which are modified in the block of code corresponding to this edge. This information is used by the \( \text{TRANS}_e \) function defined later. For each procedure \( p \) in the program, we also compute the list of variables (global variables or the formal parameters to the procedure \( p \)) modified by the procedure invocation \( S(p) \). In the absence of aliasing, this information can easily be computed by flow-insensitive interprocedural analysis in time linear to the size of call graph of the program [11]. This information is used by the \( \text{CMOD}_e \) function defined later. Similarly, for each procedure \( p \) and each candidate \( C \), we determine if the procedure invocation \( S(p) \) has any occurrence of the candidate. This information can also be computed by flow-insensitive analysis, in time proportional to the size of the call graph. This information is used by the function \( \text{OCR}_{cs} \) defined later.

5 Interprocedural Partial Redundancy Elimination

We now present the IPRE scheme we have developed. We use the terms edge and the block of code corresponding to it interchangeably in this section.

Given the full program representation we described in Section 4, the major difficulties in applying data flow analysis for PRE are:

- A procedure having a candidate for placement (or a procedure calling such a procedure) may be called at multiple call sites with different sets of actual parameters, leading to different sets of influencers. (e.g. in the code shown in Figure 4, procedure \( P \) is called at two call sites with different parameters). While considering placement of the candidate outside the procedure it is originally placed, it must be ensured that only the computation of the candidate with correct set of influencers is visible during each call of the procedure.

- For placement of a candidate at a certain point in a certain procedure, besides safety and profitability of the placement, it is also required that all influencers of the candidate are visible inside that procedure, i.e. each of them is either a global variable, a formal parameter or a local variable. (e.g. in the code shown in Figure 4, no placement will be possible inside procedure \( Q \)).

- If a procedure is called at several call sites in the program, our program representation shows paths from edges ending at a call site calling this procedure to the edges starting at other call sites for this procedure. (e.g. in Figure 5 there is a path from edge 6 to edge 9 to edge 2. Edge 6 ends at call site cs5 whereas edge 2 starts at call site cs2). These paths are never taken and the data flow analysis must not lose accuracy because of such paths in the graph.
 Transparency of blocks of code cannot be determined before starting the solution of data flow equations, since it is not known what are the local variables which need to be unmodified for the propagation of data flow information.

5.1 Lattice for Data Flow Problems

We assume that the result of the computation of a candidate is always placed in a global store, i.e. it is not passed along as an actual parameter at the call sites. Consider a procedure $p$ which has a candidate $C$ for placement and is called at call sites $cs_1$ and $cs_2$ with different sets of parameters. Our scheme cannot place this candidate at a point from which there are paths leading to $cs_1$ and $cs_2$ and these paths do not have any further computation of $C$. This restriction must be incorporated while propagating availability and while considering locations for possible placement (PPIN and PPOUT).

For this purpose, we use a three-level lattice for the determining availability and placement. The lattice is shown in Figure 6. Each middle element in the lattice refers to a list of influencers, i.e. $Infl_i = <v_1,v_2,\ldots,v_n>$. If a candidate is available or if its placement is possible, it is always with a list of influencers, which will be used in placing the candidate (i.e., if it is decided that the candidate is to be placed at this location).

We define the following functions on this lattice: $\lor$ and $\land$ are standard binary join and meet operators, i.e., for two lattice elements $a$ and $b$, $a \lor b$ is the largest element which is lower than or equal to both $a$ and $b$, and similarly, $a \land b$ is the lowest element which is greater than or equal to both $a$ and $b$. More details of data flow lattice terminology are available from standard compiler texts [2]. For ease in presenting our data flow equations, we use $\lor$ and $\land$ as confluence join and meet operators i.e. for computing join and meet respectively over a set of elements. $\neg$ is a unary operator which returns $\perp$ when applied to a list of influencers or $T$ and returns $T$ when applied to $\perp$. $\Psi$ is a binary non-commutative operators whose definition is as follows:

![Figure 6: Lattice used For Availability and Placement](image)
In intuitively, if the first operand of $\triangle$ is not $\bot$, then this function returns the first operand and if the first operand is $\bot$, it returns the second operand.

Partial availability is used only for determining profitability of placement. At any point the program, a candidate may be partially available with more than one set of influencers. i.e. there may be an occurrence of the candidate with different set of influencers along different paths leading to this point. We therefore, use a more general lattice for determining partial availability. The lattice is shown in Figure 7. The standard join and meet operators are used on this lattice.

5.2 Terminology

We further use the following terms to describe the data flow equations in this paper. We had defined our program representation earlier in Section 4. In our Full Program Representation (FPR), the entry node corresponding to the main procedure is referred to as BEGIN node in the graph and similarly, the return node corresponding to the main is referred to as the END node in the graph.

The set of procedure return nodes is represented by $R$ and the set of procedure entry nodes is represented by $E$. Consider an edge $e = (v, w)$. The source node of $e$ (i.e. the node $v$) is also referred to as $So(e)$ and the sink node of $e$ (i.e. the node $w$) is also referred to as $Si(e)$. If the sink of the edge $e$ is a procedure entry node, then the call site with which the edge $e$ is associated at its end is denoted by $cs_{\text{end}}(e)$. Alternatively, if the source of the edge $e$ is a procedure return node, then the call site with which the edge $e$ is associated at the start is denoted by $cs_{\text{beg}}(e)$. 

\begin{align*}
T \triangle x & = T \\
Infl_i \triangle x & = Infl_i \\
\bot \triangle x & = x
\end{align*}
For an edge \( e \) in the graph, we define the following sets. We will use these sets extensively in presenting the data flow equations.

\( \text{pred}(e) \): For an edge \( e = (v, w) \), \( \text{pred}(e) \) is the set of edges whose sink node is \( v \).

\( \text{succ}(e) \): For an edge \( e = (v, w) \), \( \text{succ}(e) \) is the set of edges whose source node is \( w \).

\( \text{succ}'(e) \): This set is defined only for the edges whose sink is a procedure entry node. For such an edge \( e \), \( \text{succ}'(e) \) is the set of edges which are associated with the call site \( \text{cs}_{\text{end}}(e) \) at their start.

\( \text{pred}'(e) \): This set is defined only for the edges whose source is a procedure return node. For such an edge \( e \), \( \text{pred}'(e) \) is the set of edges which are associated with the call site \( \text{cs}_{\text{beg}}(e) \) at their end.

Consider any edge \( e \) whose source is a procedure entry node. The set \( \text{cobeg}(e) \) comprises of edges whose source is the same as the source of edge \( e \). If an edge \( e \) has a procedure return node as the source and if \( cs \) is the call site with which the edge \( e \) is associated at its start, then the set \( \text{cobeg}(e) \) comprises of the edges which are associated with the call site \( cs \) at their start.

Next, consider any edge \( e \) whose sink is a procedure return node. The set \( \text{coend}(e) \) comprises of the edges whose sink is the same as the sink of the edge \( e \). If an edge \( e \) has a procedure entry node as the source and if \( cs \) is the call site with which the edge \( e \) is associated at its end, then the set \( \text{coend}(e) \) comprises of edges which are associated with the call site \( cs \) at their end.

The sets \( \text{pred}(e) \), \( \text{pred}'(e) \), \( \text{succ}(e) \), \( \text{succ}'(e) \), \( \text{cobeg}(e) \) and \( \text{coend}(e) \) for edges in the Graph shown in Figure 4 are shown in Figure 8.

At any call site \( cs \), the set of actual parameters passed is \( \text{ap}_{cs} \) and the \( j^{th} \) actual parameter is \( \text{ap}_{cs}(j) \). The set of formal parameters of the procedure called at the call site \( cs \) is \( \text{fp}_{cs} \). (Clearly, this set is the same for all call sites which call this procedure). The \( j^{th} \) formal parameter is denoted by \( \text{fp}_{cs}(j) \). The set of global variables in the program is \( gv \).

### 5.3 Availability and Partial Availability

The equations for computing availability and partial availability are given in Figure 10. In computing availability, all unknowns are initialized with \( \top \). This state means that the candidate may be available, but we do not yet know what will be the list of influencers if it is available. Bottom element in the lattice means that the candidate is not available.
\[
\begin{align*}
T_1(v_i) &= \begin{cases} 
  v_i & \text{if } v_i \in g^v \\
  f_{pcs}(j) & \text{if } v_i = a_{pcs}(j)
\end{cases} \\
\text{RNM1}_{cs}[< v_1, \ldots, v_n >] &= \begin{cases} 
  \bot & \text{if } \exists i, (v_i \notin g^v) \land (\forall j, v_i \neq a_{pcs}(j)) \\
  < T_1(v_1), \ldots, T_1(v_n) > & \text{otherwise}
\end{cases} \\
T_2(v_i) &= \begin{cases} 
  v_i & \text{if } v_i \in g^v \\
  a_{pcs}(j) & \text{if } v_i = f_{pcs}(j)
\end{cases} \\
\text{RNM2}_{cs}[< v_1, \ldots, v_n >] &= \begin{cases} 
  \bot & \text{if } \exists i, (v_i \notin g^v) \land (\forall j, v_i \neq f_{pcs}(j)) \\
  < T_2(v_1), \ldots, T_2(v_n) > & \text{otherwise}
\end{cases}
\end{align*}
\]

(5.1)

Figure 9: Renaming functions

\[
\begin{align*}
AVIN(e) &= \begin{cases} 
  \bot & \text{if } So(e) \text{ is BEGIN node} \\
  \bigwedge_{p \in pred(e)} (\text{RNM1}_{cs, end(p)}[\text{AVOUT}(p)]) & \text{if } So(e) \in \mathcal{E} \\
  \text{CMOD}_{cs, \text{beg}(e)}[\bigwedge_{p \in pred(e)} \text{AVOUT}(p)] & \text{if } (So(e) \in \mathcal{R}) \land (\neg \text{OCR}(cs, \text{beg}(e)))^3 \\
  \text{RNM2}_{cs, \text{beg}(e)}[\bigwedge_{p \in pred(e)} \text{AVOUT}(p)] & \text{if } (So(e) \in \mathcal{R}) \land (\text{OCR}(cs, \text{beg}(e)))
\end{cases}
\end{align*}
\]

(5.4)

\[
\begin{align*}
AVOUT(e) &= \bigwedge_{c \in coend(e)} (\text{TRANS}_c[\text{ANTLOC}(c) \uplus \text{AVIN}(e)]) \\
\text{PAVIN}(e) &= \begin{cases} 
  \bot & \text{if } So(e) \text{ is BEGIN node} \\
  \bigvee_{p \in pred(e)} (\text{RNM1}_{cs, end(p)}[\text{PAVOUT}(p)]) & \text{if } So(e) \in \mathcal{E} \\
  \text{CMOD}_{cs, \text{beg}(e)}[\bigvee_{p \in pred(e)} \text{PAVOUT}(p)] & \text{if } (So(e) \in \mathcal{R}) \land (\neg \text{OCR}(cs, \text{beg}(e)))^5 \\
  \text{RNM2}_{cs, \text{beg}(e)}[\bigvee_{p \in pred(e)} \text{PAVOUT}(p)] & \text{if } (So(e) \in \mathcal{R}) \land (\text{OCR}(cs, \text{beg}(e)))
\end{cases}
\end{align*}
\]

(5.5)

\[
\begin{align*}
\text{PAVOUT}(e) &= \text{TRANS}_c[\text{ANTLOC}(c) \uplus \text{PAVIN}(e)]
\end{align*}
\]

(5.6)

Figure 10: Data Flow Equations for Availability and Partial Availability

Initially, the local data flow property \(\text{ANTLOC}(i)\) of the edges in the graph is determined. (For a block of code, \(\text{ANTLOC}\) means that there is an occurrence of this candidate inside the block.) In Section 4.2, we had discussed how procedures are marked with candidates for placement. Consider an edge \(i\) whose source is a procedure entry node \(s_p\). If a candidate \(C\) is marked for placement from the procedure \(p\) with the list of influencers \(Infl_c\), we set

\[
\text{ANTLOC}_C(i) = \text{Infl}_c
\]

In all other cases, \(\text{ANTLOC}(i)\) is set to \(\bot\).

The following functions are used in our data flow equations. \(\text{TRANS}_c[Infl_i]\) of an edge \(e\) in the graph returns the list \(Infl_i\) if none of the influencers in the list \(Infl_i\) is modified in the block of code associated with this edge. If any of these influencers is modified, this function returns \(\bot\). \(\text{TRANS}_c[\top]\) and \(\text{TRANS}_c[\bot]\) are defined to be \(\top\) and \(\bot\) respectively. For a call site \(cs\) which calls procedure \(p\), \(\text{CMOD}_{cs}[Infl_i]\) returns the list \(Infl_i\) if none of the influencers in the list \(Infl_i\) is modified in the procedure invocation \(S(p)\). Otherwise \(\text{CMOD}_{cs}[\top]\) always returns \(\top\) and \(\text{CMOD}_{cs}[\bot]\)
always returns ⊥. Consider a call site $cs$ which calls procedure $p$ and a particular candidate $C$. $OCR_c(cs)$ determine if the procedure invocation $S(p)$ includes any occurrence of the candidate $C$. (Clearly, this will be the same for all call sites which call procedure $p$). Whenever there is no scope for ambiguity, we drop the subscript $C$. $OCR(cs)$ returns $\top$ or true when there is an occurrence of the candidate in the procedure invocation $S(p)$ and $\perp$ (or false) when there is no occurrence of the candidate at procedure $p$.

For renaming of formal parameters at call sites, we define two functions $RNM1_{cs}$ and $RNM2_{cs}$ (see Figure 9). Suppose a candidate is available at a call site $cs$ with a list of influencers $Inf_i$. The function $RNM1_{cs}$ determines if this candidate can be available inside the procedure called at $cs$, and if so, with what list of influencers. If any of influencers is neither a global variable nor an actual parameters at $cs$, $RNM1_{cs}$ returns $\perp$, otherwise, each actual parameter in the list is replaced by corresponding formal parameter. $RNM1_{cs}[\top]$ and $RNM1_{cs}[\perp]$ are defined to be $\top$ and $\perp$ respectively. Suppose a candidate is available at the return of a procedure and let $cs$ be one of the call sites which call this procedure. $RNM2_{cs}$ determines if this candidate will be available after the entry of the edges which start at call site $cs$. If any of the influencers of the candidate inside the procedure is neither a global variable, nor a formal parameter, then $RNM2_{cs}$ returns $\perp$. Otherwise, each formal parameter is replaced by the actual parameter at call site $cs$.

We have assumed that no aliasing is possible. This means that no global variable may be passed as an actual parameter at any call site. Similarly, no local variable or formal parameter may be passed as actual parameter more than once at any procedure call. So, the functions $RNM1_{cs}$ and $RNM2_{cs}$ always return a unique value for any input, for any call site $cs$. More over, at any call site, if $RNM1_{cs}[Inf_n]$ returns a list of influencers $Inf^*_n$ then it follows that $RNM2_{cs}[Inf^*_n]$ will return $Inf_n$. This property will be used later in establishing correctness of the system of equations.

The equations for propagation of availability can be explained as follows (see Figure 10). Consider an edge $e$ whose source is a procedure entry node. A candidate will be available at the entry of this edge $e$ if the following holds: This candidate should be available at the exit of any edge $p$ which ends at this procedure entry node (i.e. $p \in \text{pred}(e)$), and furthermore, after renaming (i.e. applying $RNM1_{cs, \text{end}(p)}$), the list of influencers with which the candidate is available should be the same for all such edges.

If an edge $e$ has a procedure return node $So(e)$ as source, $e$ is associated with call site $cs[kg(e)]$ at its start. The set $\text{pred}(e)$ comprises of edges whose sink is node $So(e)$ and the set $\text{pred}'(e)$ comprises of edges which are associated with the call site $cs[kg(e)]$ at their end. Note that even if the candidate is available at the end of all the edges $p'$ ($p' \in \text{pred}'(e)$) and none of the influencers is modified inside the procedure, the candidate may not be available inside the procedure. This can happen for two reasons, all influencers of the candidate may not be visible inside the procedure, or the procedure may be called at multiple call sites and the candidate may not be available at other call sites.

If there is no occurrence of the candidate in the procedure (CMOD_{cs[kg(e)]} does not return $\perp$), then AVIN($e$) is determined by AVOOUT at the edges belonging to $\text{pred}'(e)$. If there is any occurrence of the candidate in the procedure, then AVIN($e$) is determined by AVOOUT at the edges belonging to $\text{pred}(e)$. Note that this step preserves calling context of the procedure, i.e. accuracy in data flow analysis is not lost if a procedure is invoked at multiple call sites.

Equation 5.4 determines availability of a candidate at the end of an edge or block of code. If there is an occurrence of the candidate in the edge with list of influencers $Inf_i$, then AVOOUT is $Inf_i$; if none of the influencers is modified along this edge. If there is no occurrence inside the edge (i.e. ANTLOC is $\perp$), then the candidate is available at the exit of the edge only if it is available at the entry of the edge.
Partial availability is to suggest profitability of transformations, it does not effect correctness and safety of transformations. Partial availability does not always guarantee that redundant code motion will not occur. We have therefore, used a simple method for determining partial availability, which may not always be accurate. The inaccuracy comes because of two reasons. CMOD and TRANS functions return $\bot$ whenever one of the influencers is modified in one of the basic blocks, this basic block may not occur in all the control flow paths taken. Secondly, calling context is not always preserved in propagating partial availability information. Precise computation of partial availability can be expensive, it will require a detailed representation like Myer’s SuperGraph [34] and use of stacks and/or graph reachability for preserving calling context [35]. Our computation of partial availability still allows loop invariant code motion and redundant computation elimination. Some other optimizations which can be obtained by suppression of partially redundant computations may not be achieved because of this simple solution.

Figure 11: Data Flow Equations for Placement

\[
\begin{align*}
PP\text{OUT}(e) &= \begin{cases}
\bot & \text{if } S_i(e) \text{ is END node} \\
\land_{s \in \text{suc}(e)} \left( \text{RMN}_2 \cup_{s \in \text{suc}(s)} [\text{PPIN}(s)] \right) & \text{if } S_i(e) \in \mathcal{R} \\
\text{CMOD}_{cs \mu_{\text{end}}(e)} [\land_{s \in \text{suc}(e)} \text{PPIN}(s')] & \text{if } (S_i(e) \in \mathcal{E}) \land (\neg \text{OCR}(cs_{\text{end}}(e))) \\
\text{RMN}_1 \cup_{s \in \text{in}(e)} [\land_{s \in \text{suc}(e)} \text{PPIN}(s)] & \text{if } (S_i(e) \in \mathcal{E}) \land (\text{OCR}(cs_{\text{end}}(e)))
\end{cases} \\
\text{TEMP1}(i) &= \land_{c \in \text{cobs}(i)} (\text{ANTLOC}(c) \uplus \text{TRANS}[\text{PPOUT}(c)]) \\
\text{TEMP2}(i) &= \text{PPOUT}(i) \uplus \text{AVOUT}(i) \\
\text{PPIN}(e) &= \begin{cases}
\bot & \text{if } S_o(e) \text{ is BEGIN node} \\
\land_{p \in \text{pred}(e)} \left( \text{RMN}_1 \cup_{p \in \text{in}(p)} [\text{TEMP2}(p)] \right) & \text{if } S_o(e) \in \mathcal{E} \\
\text{PAVIN}(e) \land \text{TEMP1}(e) \land \\
\land_{p \in \text{pred}(e)} \left( \text{CMOD}_{cs \mu_{\text{end}}(e)} [\land_{s \in \text{pred}(s')} \text{TEMP2}(p')] \right) & \text{if } (S_o(e) \in \mathcal{R}) \land (\neg \text{OCR}(cs_{\text{beg}}(e))) \\
\text{PAVIN}(e) \land \text{TEMP1}(e) \land \\
\land_{p \in \text{pred}(e)} \left( \text{RMN}_2 \cup_{s \in \text{pred}(s')} \text{TEMP2}(p) \right) & \text{if } (S_o(e) \in \mathcal{R}) \land (\text{OCR}(cs_{\text{beg}}(e)))
\end{cases} \\
\text{INSERT}(e) &= \text{PPOUT}(e) \land \neg (\text{PPOUT}(e) \land \text{AVOUT}(e)) \land \\
& \quad (\neg (\text{PPIN}(e) \land \text{PPOUT}(e)) \lor \neg \text{TRANS}[\text{PPOUT}(e)]) \\
\text{DEL}(e) &= \text{ANTLOC}(e) \land \text{PPIN}(e)
\end{align*}
\]

and if none of the influencers is modified along the edge.

In computing partial availability, all unknowns are initialized with $\bot$. The equations for computing partial availability (Equations 5.5 and 5.6) are very similar to corresponding equations for computing availability, except that join operator is used instead of meet operator.
5.4 Data Flow Analysis for Placement

The data flow equations for determining placement of computations are shown in Figure 11. We briefly explain some of the key terms in these equations.

In computing PPIN in the intraprocedural case, the product term PPOUT + AVOUT ensures availability of the candidate at the entry of the node in the optimized program. PPOUT means that the candidate will be available as a result of the placements determined by the scheme. AVOUT means that the candidate is available in the original program. In the interprocedural case, the same candidate can be available with more than one list of influencers. In computing PPIN in the interprocedural scheme, we use the term PPOUT \& AVOUT (Equation 5.8). If PPOUT is set to a list of influencers Infli, then, after the placement determined by the scheme, the candidate will be available with set of influencers Infli, even if it is available with a different list of influencers before the optimized placements. If PPOUT is ⊥ and AVOUT is \{Inflj\}, then the candidate will be available with the same set of influencers Inflj even after the placement.

The rational behind the equation for determining INSERT is as follows. We decide to insert a candidate with the set of influencers Infli at the end of a block of code e, if PPOUT(e) is Infli, AVOUT(e) is not Infli and either PPIN(e) is not Infli or one of the influencers in the list Infli is modified in this block of code. The term \(\neg (\text{PPOUT}(e) \land \text{AVOUT}(e))\) will return \(\top\) whenever PPOUT(e) and AVOUT(e) are not set to the same list of influencers Infli.

In determining placement (PPIN and PPOUT), we preserve the calling context of the procedures by using a simple method, the same that we used for computing availability. It can be shown that the safety of placement is maintained through this method.

The initial value of the PPIN and PPOUT are set to ⊥. The solution can be found by iterative method.

5.5 Final Local Placement

We have so far considered the block of code associated with a single edge of FPR as the unit of placement. The final placement of the candidates which have to be inserted depends upon further intraprocedural analysis and is not necessarily at the end of blocks of code.

Consider an edge \(e_6\) for which INSERT\((e_6)\) is Infli. If this edge has a procedure return node \(r_p\) as sink, we denote by set \(S\) all the edges which have the return node \(r_p\) as the sink. If this edge has a procedure entry node as sink, let cs be the call site at which this edge ends and then we denote by \(S\) all the edges which end at call site cs.

All edges in the set \(S\) have the same succ and succ' sets, so they all have the same value of PPOUT. Therefore, the value of INSERT for any such edge can be either Infli or ⊥. Let \(I\) be the set of edges which are in \(S\) and for which INSERT\((e)\) is Infli. Let \(N\) be the set of edges which are in \(S\) and for which INSERT\((e)\) is ⊥.

It can be easily seen that all edges \(e\) in set \(S\) will have the same value of PPOUT\((e)\) and AVOUT\((e)\). Therefore, the difference in the value of INSERT\((e)\) comes because of the difference in the value of PPIN\((e)\) and TRANS\(_\varepsilon\). The following intraprocedural analysis is done for determining final placement. From the call site cs or the procedure return statement, the control flow graph is traversed backwards. Along any such traversal path, we identify the first basic block which belongs to at least one block of code in the set \(I\) but not belong to any block of code in the set \(N\). Let \(bb_i\) be such a basic block and
<table>
<thead>
<tr>
<th>Edge</th>
<th>AVIN</th>
<th>AVOUT</th>
<th>PAVIN</th>
<th>PAVOUT</th>
<th>PPIN</th>
<th>POUT</th>
<th>PPIN</th>
<th>DEL</th>
<th>INSERT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>&lt;a,b&gt;</td>
<td>↓</td>
<td>↓</td>
<td>&lt;a,b&gt;</td>
</tr>
<tr>
<td>2</td>
<td>&lt;a,b&gt;</td>
<td>↓</td>
<td>&lt;a,b&gt;</td>
<td>&lt;a,b&gt;</td>
<td>↓</td>
<td>&lt;a,b&gt;</td>
<td>↓</td>
<td>↓</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>&lt;a,b&gt;</td>
<td>&lt;a,b&gt;</td>
<td>&lt;a,b&gt;</td>
<td>&lt;a,b&gt;</td>
<td>↓</td>
<td>&lt;a,b&gt;</td>
<td>↓</td>
<td>↓</td>
<td>&lt;a,c&gt;</td>
</tr>
<tr>
<td>4</td>
<td>↓</td>
<td>↓</td>
<td>&lt;x,y&gt;</td>
<td>&lt;x,y&gt;</td>
<td>↓</td>
<td>&lt;x,y&gt;</td>
<td>↓</td>
<td>↓</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>&lt;a,b&gt;</td>
<td>&lt;a,b&gt;</td>
<td>&lt;a,b&gt;</td>
<td>&lt;a,b&gt;</td>
<td>↓</td>
<td>&lt;a,b&gt;</td>
<td>↓</td>
<td>↓</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>&lt;a,c&gt;</td>
<td>&lt;a,c&gt;</td>
<td>&lt;a,c&gt;</td>
<td>&lt;a,c&gt;</td>
<td>↓</td>
<td>&lt;a,c&gt;</td>
<td>↓</td>
<td>↓</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>&lt;a,b&gt;</td>
<td>&lt;a,b&gt;</td>
<td>&lt;a,b&gt;</td>
<td>&lt;a,b&gt;</td>
<td>↓</td>
<td>&lt;a,b&gt;</td>
<td>↓</td>
<td>↓</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>&lt;a,c&gt;</td>
<td>↓</td>
<td>↓</td>
<td>&lt;a,c&gt;</td>
</tr>
<tr>
<td>9</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>&lt;a,c&gt;</td>
<td>↓</td>
<td>↓</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>&lt;a,c&gt;</td>
<td>↓</td>
<td>&lt;a,c&gt;</td>
<td>&lt;a,c&gt;</td>
<td>↓</td>
<td>&lt;a,c&gt;</td>
<td>↓</td>
<td>↓</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>↓</td>
<td>&lt;y,z&gt;</td>
<td>&lt;y,z&gt;</td>
<td>&lt;y,z&gt;</td>
<td>↓</td>
<td>&lt;y,z&gt;</td>
<td>↓</td>
<td>↓</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>&lt;a,c&gt;</td>
<td>&lt;a,c&gt;</td>
<td>&lt;a,c&gt;</td>
<td>&lt;a,c&gt;</td>
<td>↓</td>
<td>&lt;a,c&gt;</td>
<td>↓</td>
<td>↓</td>
<td></td>
</tr>
</tbody>
</table>

Figure 12: Solution of Data Flow Properties for the Graph

Program Foo

```plaintext
a = 1
Sched(a, b)
Do i = 1, 100
    Call P(a, b)
    Call Q(c)
Enddo
Call Q(c)
Sched(a, c)
Call P(a, c)
If cond then
    Call Q(a)
    Sched(a, c)
Endif
Call R(a, c)
End
```

Procedure P(x, y)

```plaintext
...other computations...
End
```

Procedure Q(z)

```plaintext
z = ...z...
End
```

Procedure R(y, z)

```plaintext
...other computations...
End
```

Figure 13: Optimized Version of Program. Note that further Intraprocedural Analysis is required at call sites cs1 and cs6 to determine final placement.

let bb_{i+1} be its successor which belongs to at least one block of code in the set \( \mathcal{N} \). We then check if an occurrence of the candidate is already available at the end of the basic block bb_{i}. If this is not the case, a new basic block is inserted between the basic blocks bb_{i} and bb_{i+1} and the candidate is inserted in this new basic block.

The solution of data flow properties for the program shown in Figure 4 is shown in Figure 12. The optimized program is shown in Figure 13.

5.6 Discussion

We now explain an important limitation of our method, and how a simple extension to our scheme can overcome this limitation. This limitation comes from how we initialize the blocks of codes with the candidates placed in them. As we explained earlier, an occurrence of a pure function is considered for hoisting out of the procedure only if none of the parameters of the pure function is modified along
any path from the start of the procedure to this invocation of the pure function and the invocation of pure function is not enclosed in any loop or conditional. This obviously can impose limitations on the effectiveness of the scheme, as a candidate enclosed in a conditional will not be considered for placement.

However, we can improve the effectiveness of the scheme by this simple extension. If a candidate with a high cost is present enclosed in a conditional, then we can insert a new node in the graph before the point where the candidate is present. Obviously, if this is done too frequently, then the size of the graph can become very large. However, we believe that a complex interprocedural framework e should only be applied for high cost candidates (like communication and communication processing constructs for distributed memory compilation), and such constructs are not present too frequently in the original source code.

In presenting our technique, we have used a novel representation for the full program, and then we have developed the technique on this representation. However, most of the details of the presented technique are applicable even if the underlying program representation is a different one, e.g., if Myer’s Supergraph is used. Specifically, our methods of preserving calling context and handling multiple invocations to the procedure having candidates are general, and apply for any interprocedural code placement problem, irrespective of the program representation used.

In our work, we have chosen the Dhamdhere’s Modified Morel Renvoise (MMRA) technique as the basis for developing interprocedural scheme. Another interesting candidate would have been Knoop et al.’s recent technique for PRE [30]. The main advantage of their technique (when applied intra-procedurally) is that it uses two successive unidirectional data flow solutions, whereas MMRA scheme uses bidirectional analysis. One likely challenge in applying Knoop’s technique will be that interprocedural refinement will be required twice, i.e. once after solving the first system of data flow equations, and then after the second system, to do the final placement. However, we cannot conclude the complexity of the analysis and difficulty of the proofs without working through all the details. Developing an interprocedural version of Knoop’s PRE scheme using our program representation will be an interesting topic for future work.

6 Theoretical Analysis and Complexity

In this section, we give an overview of the correctness and safety of the set of data flow equations. We also analyze the complexity of the method here. The detailed proofs are given in the accompanying technical report.

The basic outline of our proof is similar to the proof of correctness given by Morel and Renvoise for the interprocedural PRE [32]. However, there are many additional complications that one needs to deal with in the interprocedural case, these can be summarized as follows:

- Whenever any occurrence of a candidate is deleted, we need to make sure that not only an occurrence of the candidate will be available, but an occurrence of the candidate with the correct set of influencers will be available.

- To maintain accuracy of data flow analysis when a procedure is called at multiple call sites, our data flow equations take into account procedures whose invocation does not include any occurrence of the candidate and does not modify any of the influencers. Thus, the propagation of data flow information goes beyond just the predecessors and successors of a given node or edge.
Our method is separated into an interprocedural phase on a concise representation and an intraprocedural phase for determining final placement. We need to establish correctness and safety of placement with this two phase methodology.

The details of how our scheme works correctly in presence of the above three features have been presented in a technical report. We have also shown that the data flow equations for determining placement converge.

### 6.1 Complexity of the Method

We now analyze the complexity of the proposed method. We first introduce the terminology and symbols we use for determining the complexity of our method.

- $N_e$: Number of blocks of code (edges) in the graph
- $N_{cs}$: Number of call sites in the program
- $N_{ps}$: Max. number of pred. / succ. of any edge
- $N_{gv}$: Number of global variables in the program
- $N_{lv}$: Max. of the no. of local variables in any procedure
- $N_{fp}$: Max. of the no. of formal parameters for any procedure
- $N_{inf}$: Number of influencers of the candidate

Since the scheme needs to be applied for each of the candidates, we analyze the cost for determining placement of one candidate here. The cost of the applying the scheme can be broken into the following components:

1. The cost of determining transparency of each of the block of code
2. The cost of evaluating the summary function for each call site
3. Applying the renaming functions at each of the call sites
4. Comparing the list of influencers after applying the transfer function from the predecessors/successors
5. Performing the analysis iteratively till the solution converges

It can be shown that the set of data flow equations satisfy the monotonicity property, i.e., any step in the data flow solution process can only result in lowering the value of a particular data flow term. Therefore, any data flow term can only have one list of influencers as the value during the solution process. Therefore, the transparency, summary functions and the renaming functions need to evaluated only once for any call site or block of code, and not during every iteration in the data flow solution process.

We can summarize the cost of each of the five components described above as follows: Transparency needs to be evaluated for each block of code. If the mod information is available for all the local and global variables, the cost of computing transparency of each block of code is $O((N_{gv} + N_{lv}) \times N_{inf})$. So, the overall cost of this first component is

$$O((N_{gv} + N_{lv}) \times N_{inf} \times N_e)$$
Similarly, the cost of evaluating the summary functions is

\[ O((N_{gv} + N_{fp}) \times N_{inf} \times N_{cs}) \]

The cost of applying the renaming functions at each of the call site is

\[ O(N_{fp} \times N_{inf} \times N_{cs}) \]

Comparing the list of influencers of the successors or predecessors of any block of code (to determine if they are equivalent) can be done in

\[ O(N_{ps} \times N_{inf} \times N_c) \]

After we have accounted the cost of these four components, the remaining cost of evaluating the data flow terms till they converge is the same as for the intraprocedural algorithm [15], which will be \( O(N_e^2) \).

To compute the overall cost, we make the following approximations: We denote by \( N_v \), the maximum number of variables visible in any part of the code. Also, \( O(N_{ea}) \) is bounded by the \( O(N_e) \). So, the overall complexity of the method is

\[ O(N_e^2 + N_e \times N_{inf} \times (N_v + N_{ps})) \]

## 7 Experimental Results

We now present experimental results to show the usefulness of the Interprocedural Partial Redundancy Elimination in distributed memory compilation. We measure the difference made by performing interprocedural placement of both the communication preprocessing statements and the collective communication statements. We have used two irregular codes in our study, an Euler solver on an unstructured mesh [13], originally developed at ICASE by Mavriplis et al. and a template taken from CHARMM [6], a molecular dynamics code. We used Intel Paragon at Rice University for performing our experiments.

The Euler solver we experimented with sweeps over an unstructured mesh inside the time step loop. The data parallel loops iterate over both the edges and the faces of the unstructured mesh. Indirection arrays are used to store the nodes corresponding to each edge and each face of the mesh. This leads to irregular accesses to data in the major computational loops of the program. The version of the code we worked with comprised of nearly 2000 lines of code across 8 procedures. We used two sets of input data in our experiments, a mesh having 33000 mesh points and 350000 edges, and another mesh having 9300 mesh points and 55000 edges.

The existing Fortran D compiler inserts appropriate communication preprocessing statements and collective communication statements in parallelizing such irregular codes, but (before the work presented here) did not perform any interprocedural placement of these statements.

In Figure 14, we show the performance difference obtained by interprocedural placements of communication preprocessing statements and communication statements. Performance of the different versions of the code is measured for 2 to 32 processors of Intel Paragon. The sequential program took 71 seconds on a single processor of Intel Paragon. A super-linear speed up was noticed in going from one processor to two processors, we believe happens because on single processor, all data cannot fit in the main memory of the machine. The first version (V 1) is the code which does not perform any interprocedural placement. In the second version (V 2), interprocedural placement is performed for only communication preprocessing statements. This leads to significant difference in the performance. Third version (V 3) is further
optimized by placement optimizations on communication statements. On small number of processors, the total communication time is small, and therefore, the overall performance difference due to the different communication optimizations is not significant. However, when the same data is distributed over a larger number of processors, the communication time becomes a significant part of the total execution time and the communication optimizations make significant difference in the overall performance of the program.

In Figure 15, we show the result of optimizations when this program is run on a larger data set, i.e. a 53000 node mesh. Interprocedural placement of communication preprocessing statements results in significant reduction in the time required by the program. When the number of processors is large, the communication time becomes significant in total execution time of the program and interprocedural optimizations on communication statements also lead to substantial improvement in the performance of the code.

The second code we considered was a template taken a molecular dynamics code Charmm [6, 28]. We used data from water molecules, which comprised of 648 atoms and nearly 100K interactions between the atoms.

In Figure 16, we show the result of optimizations. The sequential program took 34.8 seconds on a single processor of Intel Paragon. In the first version (V 1), no interprocedural placement of communication preprocessing statements is done. In the second version (V 2), placement of communication preprocessing statements is optimized interprocedurally. Since this was a relatively small template, no further improvement in performance can be achieved by interprocedural optimization of communication

Figure 14: Effect of Optimizations on Euler solver (10K mesh, 20 iterations) on Intel Paragon.
Figure 15: Effect of Optimizations on Euler solver (53K mesh, 20 iterations) on Intel Paragon.

7.1 Discussion

We now comment on the effectiveness and efficiency of our implementation of IPRE. As we mentioned earlier, we are using a concise representation for our analysis, which can miss certain optimizations that a more detailed representation like Myer’s Supergraph will allow. So, one interesting question is the effectiveness of our technique, on the real codes used for the study. In the real codes used for our experiments, our IPRE scheme could find all the optimizations that would have been possible even with a detailed representation. Thus, FPR turned out to be as effective for optimizations as Myer’s Supergraph.

The next question is how the efficiency of the analysis was improved by the use of a concise representation. This question cannot be completely answered unless a complete implementation of IPRE is done using both FPR and Supergraph. Such a detailed comparison will involve significant new effort and is beyond the scope of this paper. However, we believe that relative sizes of call graph, FPR and Supergraph will illustrate the relative efficiency of the use of these representations. For the Euler code used for our first set of experiments, the call graph has 8 nodes (since the code has 8 procedures). FPR had 19 blocks of code, whereas the total number of nodes in the Supergraph is 226. For the second code, the call graph had 2 nodes, the FPR had 4 blocks of code and the Supergraph had 23 nodes. This indicates that the efficiency of the analysis using the FPR is close to the efficiency of flow-insensitive analysis using the call graph, and is significantly lower than the cost of analysis with the Supergraph.
Figure 16: Effect of Optimizations on Charmm template (20 iterations) on Intel Paragon.

8 Related Work

We are aware of two efforts on performing interprocedural partial redundancy elimination. Morel and Renvoise briefly discuss how their scheme can be extended interprocedurally [33]. Their solution is heuristic in nature, and no formal details are available for their interprocedural scheme. Their work is restricted to the programs whose call graph is acyclic. They also do not consider the possibility that the procedure having a candidate for placement may be invoked at multiple call sites with different set of parameters and do not maintain accuracy of solutions when procedures are invoked at multiple call sites.

Knoop et al. extend a scheme for performing earliest possible code motion interprocedurally [29]. The main limitation of their work is that if any of the influencers of a candidate is a formal parameter, then the candidate is not considered for placement outside procedure boundary (since no renaming of influencers is done). In the example presented in this paper, as well as in the Euler code we used for our experiments, their scheme will not perform any code motion. They do not consider the possibility of using any concise representation for the full program. Also, we believe that our effort is the first one to report an implementation and application of interprocedural partial redundancy elimination.

Partial Redundancy Elimination encompasses loop invariant code motion and common subexpression elimination. The only effort on loop invariant code motion that we are familiar is of Hall et al. [21]. They use a program abstraction Augmented Call Graph (ACG) which explicitly stores information about loops enclosing a procedural call. Their work is restricted to considering only the programs whose call graph is a directed acyclic graph (DAG).
We briefly compare our work with efforts on other flow-sensitive interprocedural problems. Several different program representations have been used for different flow-sensitive interprocedural problems. Myer has suggested concept of SuperGraph [34] which is constructed by linking control flow graphs of subroutines by inserting edges from call site in the caller to start node in callee. The total number of nodes in SuperGraph can get very large and consequently the solution may take much longer time to converge. Several ideas in the design of our representation are similar to the ideas used in Callahan's Program Summary Graph [9] and Interprocedural Flow Graph used by Soffa et al. [24]. FIAT has been introduced as a general framework for performing interprocedural analysis [20], but is more targeted towards flow-insensitive problems. A number of data flow problems have been solved interprocedurally. This includes analysis for determining use and kill of variables [34, 4, 5, 7, 9, 12], determining use-definition and definition-use chains [24], computing interprocedural slices [27], and interprocedural constant propagation [10]. Interval based approach for solving interprocedural data flow equations has been investigated by Burke [7] and recompilation in a compiler performing interprocedural analysis has been investigated by Burke and Torczon [8].

An obvious alternative to interprocedural analysis is inline substitution [36]. In this method, the call sites are replaced by body of the callee procedure. Subsequently, standard techniques for intraprocedural analysis can be applied for optimizing. We believe that, for several reasons, inlining is not an acceptable alternative for the kind of optimizations we are performing here. Firstly, inlining can increase the size of the code exponentially, and therefore, cause high overheads. Secondly, inlining is not applicable to recursive routines. Inlining has been successful and profitable only when applied very selectively, e.g., a common heuristic is to inline only the procedures which are non-recursive, do not call any procedure and are called only once in the entire program body. Such limited inlining cannot allow the optimizations performed here.

Yet another alternative is to create a separate version of each procedure for each call site. This method is known as procedure cloning. Once all the procedures have been cloned, the interprocedural analysis can get relatively simple. However, this approach has the same drawbacks as inline substitution, i.e. code explosion. To avoid this problem, cloning should be applied selectively. Our analysis can be augmented to determine where cloning a procedure may improve the applicability of optimizations.

9 Conclusions

In this paper we have addressed the problem of performing partial redundancy elimination interprocedurally. We have developed an interprocedural partial redundancy elimination (IPRE) scheme. Our method is applicable on arbitrary recursive programs and arbitrary control flow within each procedure.

We developed a new concise full program representation for performing data flow analysis. We have shown that our interprocedural scheme maintains correctness and safety when applied on this concise representation. For preserving calling contexts in propagating data flow information, we compute side-effects of each procedure. We have shown how our analysis maintains correctness and safety with this simple and efficient method for preserving calling context of procedures.

We use interprocedural partial redundancy elimination for determining optimized placement of communication preprocessing and communication statements while compiling for distributed memory machines. We have implemented our scheme as an extension to the existing Fortran D compilation system. Experimental results from two irregular codes compiled using our system show how interprocedural
placement of these statements is a must for obtaining reasonable performance. We have not been able
to perform any experiments with regular codes to demonstrate the applicability of interprocedural place-
ment. Since the cost of communication and communication preprocessing is usually lower for regular
applications, we expect relatively lower performance gains, and on only a limited fraction of such appli-
cations. More detailed study of the applicability of interprocedural placement for regular applications is
a subject for future study.

Acknowledgments

This research would not have been possible without the support and encouragement of Joel Saltz. We
have implemented this scheme as an extension to the D system being developed under the leadership
of Ken Kennedy at Rice University. We gratefully acknowledge our debt to the implementers of the
interprocedural infrastructure (FIAT) and the existing Fortran D compiler. We are also grateful to the
members of the CHAOS team at University of Maryland for their continuous help with the use of CHAOS
software during our experiments. The research of Agrawal was supported in part by NSF CAREER award
from Advance Scientific Computing division.

References

[1] Gagan Agrawal, Alan Sussman, and Joel Saltz. Compiler and runtime support for structured and block
November 1993.

Wesley, 1986.

[3] Bhavin Avalani, Ian Foster, Ming Xu, and Alok Choudhary. A compilation system that integrates HPF
and Fortran M. In *Proceedings of the Scalable High Performance Computing Conference (SHPCC-94)*, pages

August 1978.

posium on the Principles of Programming Languages*, Los Angeles, January 1977.

program for macromolecular energy, minimization, and dynamics calculations. *Journal of Computational


[9] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Pro-
cedings of the SIGPLAN ‘88 Conference on Programming Language Design and Implementation*, Atlanta, GA,

[10] D. Callahan, K. Cooper, K. Kennedy, and L. Torczon. Interprocedural constant propagation. In *Pro-


m programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):491–523,
October 1986.


Mary Jean Harrold and Mary Lou Soafa. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, March 1994.


