Distribution Assignment Placement

Eduard Mehofer Hans P. Zima

Institute for Software Technology and Parallel Systems
University of Vienna, Liechtensteinstr. 22, A-1090 Vienna, Austria
e-mail: {mehofer,zima}@par.univie.ac.at

Abstract

The change of distributions of arrays at runtime in languages such as High Performance Fortran (HPF) addresses the demands posed by advanced applications with dynamically varying processor workloads or varying computational kernels. We introduce distribution assignments which are generated by the compiler at subprogram boundaries and for redistribute/realign directives in order to enforce distribution changes. Distribution assignments serve as intermediate representation which allows an abstraction from high-level language constructs which simplifies the analysis task and permits a generic description of the optimization phase. In this paper an aggressive optimization based on bidirectional data flow frameworks is presented which reduces the number of executed distribution assignments more effectively than usual approaches. Instead of implementing individual optimizations, a more general problem, namely elimination of partially dead and partially redundant distribution assignments, is solved which results in a uniform framework covering various optimizing transformations. Unlike many communication optimization systems, code hoisting does not suffice and has to be complemented by code sinking.

1 Introduction

In order to make programming of distributed-memory multiprocessing systems easier, high-level languages such as HPF [6], Fortran D [7], and Vienna Fortran [18] have been proposed which enable users to write programs in much the same way as they are accustomed to on sequential machines. The similar way of programming is achieved by offering a global address space, providing language support for distributing data, and assuming a single thread of control.

The main feature of those data-placement languages is the user-controlled distribution of data across the local memories of the processing nodes. Since the distribution of data has a crucial impact on runtime performance, a change of a distribution may be necessary in order to prevent workload imbalance or bad data locality for varying computational phases. Therefore, HPF allows to modify the association of an array with a distribution at subprogram boundaries and with redistribute/realign directives. We use distribution assignments to enforce an association of a distribution with an array. Distribution assignments are generated by the compiler and serve as intermediate representation which allows an abstraction from high-level language constructs and permits a generic description of the optimization phase.

Distribution assignments can be very expensive operations. Communication must be generated to migrate the array elements to the processing nodes which will be the new owners. Therefore, it is a primary goal to reduce the number of executed remappings. In [9], individual optimizations have been developed for eliminating and moving remappings separately. But both optimizations can be performed at once by solving a more general problem which is based on partial redundancy elimination (PRE) [16] and partial dead code elimination (PDE) [12] techniques, respectively, resulting in more powerful transformations. A distribution assignment is called *partially redundant* iff there exists at least one execution path where the assignment is *redundant* and thus executed twice on that path. Analogously, a distribution assignment is called *partially dead* iff there exists at least one execution path where the assignment is *dead* and thus superfluous on that path.

Conceptually, partial redundancy elimination of distribution assignments (PRED) consists of hoisting and eliminating individual distribution assignments. The principle of PRED is to move assignments against the direction of the control flow and to place them as *universal* as possible. In this way, the potential of redundant assignments is maximized, which can be deleted thereafter. Unlike many communication optimization systems [8, 1], code hoisting does not suffice and has to be complemented by code sinking as shown in Section 2. Analogously to PRED, partial dead code elimination of distribution assignments (PDED) consists conceptually of sinking and eliminating individual distribution assignments. The principle of PDED is to move assignments in the direction of the control flow and to place them as *specific* as possible. In this way, the potential of dead assignments is maximized.

In this paper a distribution assignment placement framework is presented which reduces the number of executed assignments more effectively than usual approaches by integrating both PRED and PDED. In order to increase the profitability of the optimization, PDED has to be performed prior to PRED. Optimizations covered by the framework in a uniform manner include elimination of redundant and dead distribution assignments, motion of loop-invariant distribution assignments just before and just after the loop, and suppression of partially redundant and partially dead distribution assignments. Since HPF and related languages allow to realign arrays dynamically, the set of arrays which have to be remapped at a redistribute directive depends on the control flow. We support dynamically realigned arrays by resolving the set of possibly aligned arrays explicitly with conditional distribution assignments. The framework is currently being implemented within the VFCS system [3].

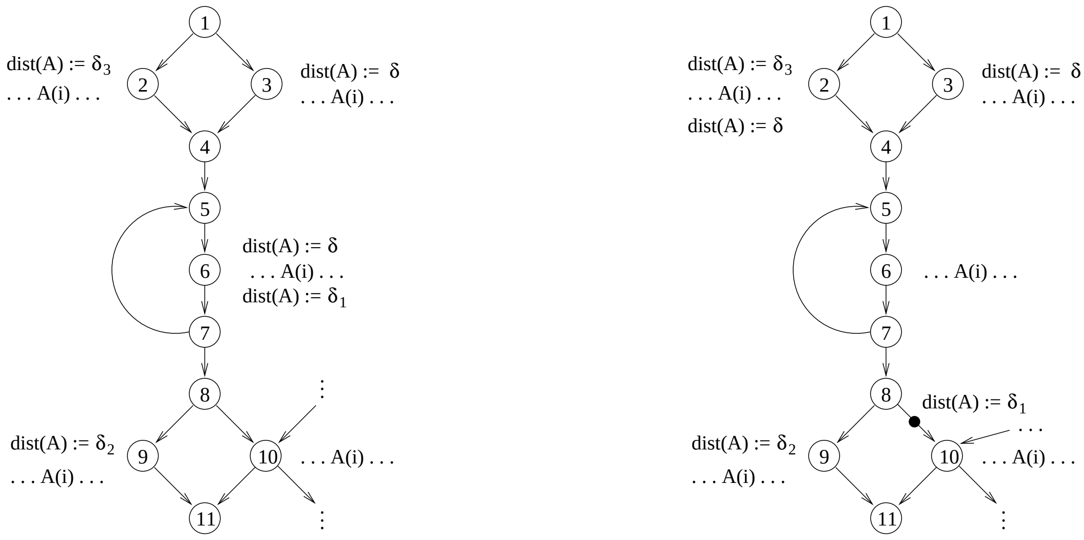
The paper is organized as follows. In Section 2 the importance of eliminating partially redundant and partially dead distribution assignments is shown. Section 3 introduces the terminology and preliminaries used throughout this paper. The translation of high-level language constructs into distribution assignments as well as alignment problems are discussed in Section 4. Section 5 and 6 present the PDED and PRED framework, respectively. The distribution placement algorithm is described in Section 7 and applied to an example in Section 8. Experimental results are presented in Section 9. The paper concludes with a discussion of related work.

2 Motivation

As already mentioned, the number of executed distribution assignments has a crucial impact on the runtime performance of a parallel program. Elimination of redundant and dead distribution assignments does not suffice as shown in Figure 1. None of the distribution assignments are redundant or dead. But the second distribution assignment at node 6 is partially dead, since it is live with respect to the right branch of the following if-statement and dead otherwise. The first distribution assignment of node 6 is partially redundant, since it is redundant with respect to the right branch of the preceding if-statement and not redundant otherwise. Note that a loop body with remappings as shown in Figure 1 is very likely to occur when subprograms with explicitly distributed arguments are called within a loop. In that case, distribution assignments are inserted implicitly by the compiler.

Since the first distribution assignment at node 6 is required to restore the distribution which is changed by the second one, we fail to hoist the assignment out of the loop. The second distribution assignment of node 6, however, is dead within the loop and, hence, we are allowed to move it just after the loop. But at node 8 the assignment is still partially dead. Since it is not safe to move the assignment to node 10, edge placement succeeds in eliminating the partially dead assignment by inserting a synthetic node on the edge $(8, 10)$.

Fortunately, the sinking of the distribution assignment from node 6 to the edge $(8, 10)$ turned



(a) Unoptimized flow graph

(b) Optimized flow graph

Figure 1: Distribution assignment motion.

the first assignment at node 6 to be redundant within the loop. Hence, we are allowed to hoist the assignment from node 6 to node 4. Since there is still a partial redundancy due to node 3, we eliminate this by hoisting the assignment farther to the exit of node 2. Now, the optimized version of the flow graph of Figure 1 does not contain partially redundant or partially dead distribution assignments any more.

Runtime techniques to reduce the number of executed redistributions (e.g. conditional remappings) would fail for the flow graph of Figure 1. If the placement of the distribution assignments is not modified at compile-time, we are forced to perform both remappings in each iteration in order to establish the requested distributions.

3 Preliminaries

The *data space* of a program P is the set of all data objects declared in P and is denoted by $\mathcal{A}$. A distribution δ of an array $A \in \mathcal{A}$ maps each array element of A to one or more processors, which become the owners of the element, and, in this capacity, store the element in their local memory. A distribution δ is defined by the tuple $(\mathbf{I}^A, \mathbf{I}^R, fcn)$ or by an alignment specification α. $\mathbf{I}^A$ and $\mathbf{I}^R$ denote the array index domain and the processor array index domain, respectively, and fcn denotes a function which maps $\mathbf{I}^A$ to $\mathbf{I}^R$. An alignment specification α is a mapping from an index domain $\mathbf{I}^A$ to an index domain $\mathbf{I}^B$ whereby A denotes an aligned array and B the alignee and the elements of A reside at the same processing node where the corresponding elements of array B are located. A distribution assignment $dist(A) := \delta$ creates an association between array A and distribution δ. Distribution assignments are generated by the compiler at subprogram boundaries and for redistribute/realign directives.

A program flow graph G of a program P is a triple $G = (N, E, s)$, where (N, E) is a directed graph, $s \in N$ is the unique entry node for program P, and there is a path from s to every node of G. Each node of the graph represents a basic block, and edges indicate transfer of control between basic blocks. The set of predecessors and successors of node i are denoted by $pred(i)$ and $succ(i)$, respectively. A path of length $k-1$ is a sequence $[x_1, \ldots, x_k]$ of nodes, such that $(x_i, x_{i+1}) \in E$ for $1 \leq i \leq k-1$. A path without starting and ending node is denoted by $(x_1, \ldots, x_k)$. A special case is a path of length 0 which consists of a single node.

4 Insertion of Distribution Assignments

A distribution assignment of the form $dist(A) := \delta$ associates distribution δ with the array A on the left. Distribution assignments are inserted by the compiler whenever an association of a distribution with an array is requested by the program. Hence, distribution assignments serve as intermediate representation which allows an abstraction from high-level language constructs and permits a language-independent description of the optimization framework.

Since HPF and related languages allow to realign arrays dynamically at statement level, the set of arrays which have to be remapped at a redistribute directive depends on the control flow. Therefore, a compiler cannot determine in general the set of aligned arrays exactly but only a superset of possibly aligned arrays. Conditional distribution assignments are generated to denote potential remappings. Since it is not expected that dynamic realignments are used very extensively, a conservative approximation based on flow-insensitive information will yield reasonable results in most cases. Consider Example 1. Arrays B and C are dynamically realigned to array A depending on some condition. Therefore, conditional distribution assignments are used to indicate that possibly a remapping of arrays B or C may occur at the redistribute directive. In this way, we are able to handle dynamic realignments and make distribution analysis amenable to traditional analysis techniques. Of course, more precise sets of possibly aligned arrays can be obtained by considering the control flow and performing aligned array analysis analogously to reaching definitions [2].

Example 1 Translation of an HPF redistribute directive.

Original Code

```
      REAL A(N,N), B(N), C(N)
!HPF$ PROCESSORS P(M,M)
!HPF$ DYNAMIC :: A, B, C
        ...
      IF (···) THEN
!HPF$    REALIGN B(:) WITH A(:,1)
      ELSE
!HPF$    REALIGN C(:) WITH A(:,1)
      END IF
        ...
!HPF$ REDISTRIBUTE (CYCLIC,CYCLIC) ONTO P :: A
```

Transformed Code

```
REAL A(···), B(···), C(···)
  ...
! redistribute directive
dist(A) := ([1:N,1:N],[1:M,1:M],(CYCLIC,CYCLIC))
IF (aligned(B,A)) dist(B) := ([1:N],[1:M],(CYCLIC)) ! conditional remapping
IF (aligned(C,A)) dist(C) := ([1:N],[1:M],(CYCLIC)) ! conditional remapping
```
■

According to the current HPF-2 document [6], remappings at subprogram boundaries are visible to the caller and can be generated in either the caller or the called subprogram. We decided that the caller establishes the distributions requested by the dummy arguments of the callee. By default, copy-in/copy-out argument passing is used. However, the copy-in/copy-out mechanism is replaced by call-by-reference whenever possible. The actual argument is passed directly to the subprogram and distribution assignments are inserted just before the subprogram to set up the requested distribution and just after the subprogram to restore the original one. The translation of a subroutine call using the call-by-reference mechanism is shown in Example 2.

Example 2 Translation of an HPF subroutine call using call-by-reference.

Original Code

```
      REAL A(N)
!HPF$ PROCESSORS P(M)
!HPF$ DISTRIBUTE (CYCLIC) ONTO P :: A
      INTERFACE
        SUBROUTINE F(A)
          REAL A(:)
!HPF$     DISTRIBUTE A (BLOCK) ONTO *
        END SUBROUTINE
      END INTERFACE
      ...
      DO I=1,10
        CALL F(A)
      END DO
```

Transformed Code (unoptimized)

```
REAL A(···)

⋮

DO I=1,10
  dist(A) := ([1:N],[1:M],(BLOCK))
  CALL F(A,···)
  dist(A) := ([1:N],[1:M],(CYCLIC))
END DO
```

In HPF, alignment directives introduce interdependences between distribution assignments. Consider Example 3. Since array B is aligned with array A, the alignment creates a dependence between A and B. Hence, we are not allowed to place the distribution assignment of B prior to A which might be useful in some situations. Therefore, we replace alignment specifications by distribution tuples whenever possible as shown in Example 3 at p_2. Remaining alignment specifications may prevent some profitable optimizations.

Example 3 Eliminating alignment dependences.

Original Code

```
      REAL A(N,N), B(N)
!HPF$ PROCESSORS P(M)
!HPF$ DYNAMIC :: A, B
      ...
!HPF$ REDISTRIBUTE A(BLOCK,BLOCK) ONTO P
!HPF$ REALIGN B(:) WITH A(:,1)
       ...
```

Transformed Code

```
REAL A(···), B(···)
...
! redistribute directive
p1 :dist(A) := ([1:N,1:N],[1:M,1:M],
               (BLOCK,BLOCK))
! realign directive
p2 :dist(B) := ([1:N],[1:M],(BLOCK))
...
```

If an initial distribution is assigned to an array or array kill analysis detects that all array elements are re-initialized after the remapping, the communication can be omitted (Vienna Fortran provides language support for detecting dead communication). Otherwise, communication is required in order to migrate the array elements to the processing nodes which will be the new owners.

5 Elimination of Partially Dead Distribution Assignments

The elimination of partially dead distribution assignments is defined dual to classical partial redundancy elimination systems. The framework presented here adapts the *edge placement* technique [5] to partial dead code elimination. A distribution assignment is placed along an edge ending in a node for which sinking is possible only for some but not all of its predecessor nodes. As shown in Section 2, the insertion of such *synthetic nodes* helps to increase the effectiveness of partial dead code elimination.

Table 1 describes the data flow sets and equations of the PDED framework which performs dead distribution assignment elimination, motion of loop-invariant distribution assignments just after the loop, and suppression of partially dead distribution assignments in a uniform manner. The array-distribution association set ADA is composed of pairs which are derived from distribution assignments and describe associations between arrays and distributions:

$ADA := \{\ (A, \delta)$: there is a distribution assignment of the form $dist(A) := \delta$ in $P\ \}$.

The data flow information set is the powerset of ADA, denoted by $\mathcal{P}(ADA)$. The local sets, which can be determined by a simple scan over the basic blocks, describe downwards exposed distribution assignments ($POSTLOC(n)$), upwards exposed distribution assignments ($GEN^S(n)$), and transparent distribution assignments ($TRANSP^S(n)$) for each node n.

Table 1: PDED Framework

Local Sets

$POSTLOC(n)$	:=	$\{\ (A,\delta) \in ADA$: there is an occurrence of $dist(A) := \delta$ in n which is not followed by any other occurrence of A nor any definition of a variable used in $\delta\ \}$
$GEN^S(n)$	:=	$\{\ (A,\delta) \in ADA$: there is an occurrence of $dist(A) := \delta$ in n prior to any occurrence of A or definition of a variable used in $\delta\ \}$
$TRANSP^S(n)$	:=	$\{\ (A,\delta) \in ADA$: there is no occurrence of A nor definition of a variable used in δ in $n\ \}$

Data Flow Equations for Eliminating Partially Dead Distribution Assignments

$PPIN^S(n)$	:=	$\bigcap_{p \in pred(n)}(PPOUT^S(p))$
$PPOUT^S(n)$	:=	$(POSTLOC(n) \cup (TRANSP^S(n) \cap PPIN^S(n))) \cap PDDOUT(n) \cap (\bigcup_{p \in succ(n)}((PPOUT^S(p) - POSTLOC(p)) \cup DDIN(p)))$
$INSERT^S(n)$	:=	$(PPIN^S(n) - DDIN(n)) - (PPOUT^S(n) \cap TRANSP^S(n))$
$INSSYNTH^S(m,n)$	:=	$PPOUT^S(m) - (PPIN^S(n) \cup DDIN(n))$
$DEAD(n)$	:=	$POSTLOC(n) \cap PPOUT^S(n)$

A distribution of an array A is called *dead* at a program point p iff on every path starting from p each access to A is preceded by a distribution assignment to A. A distribution assignment is identified as *dead code* iff the distribution is *dead*. The global sets $DDIN(n)/DDOUT(n)$ and

$PDDIN(n)/PDDOUT(n)$ denote distribution assignments which are dead or partially dead at entry/exit of node n, respectively:

$$\begin{array}{lll} DDIN(n) & := & \{(A,\delta) \in ADA : \text{distribution of } A \text{ is dead at entry of node } n\ \} \\ DDOUT(n) & := & \{(A,\delta) \in ADA : \text{distribution of } A \text{ is dead at exit of node } n\ \} \\ PDDIN(n) & := & \{(A,\delta) \in ADA : \text{distribution of } A \text{ is partially dead at entry of node } n\ \} \\ PDDOUT(n) & := & \{(A,\delta) \in ADA : \text{distribution of } A \text{ is partially dead at exit of node } n\ \} \end{array}$$

The sets $DDIN$ and $DDOUT$ are obtained by $DDIN(n) = GEN^S(n) \cup (TRANSP^S(n) \cap DDOUT(n))$ and $DDOUT(n) = \bigcap_{p \in succ(n)}(DDIN(p))$. Partial dead distribution analysis differs from dead distribution analysis only in the confluence operator which is changed from intersection $\bigcap$ to union $\bigcup$.

The $PPOUT^S$ equation consists of the safety term $POSTLOC(n) \cup (TRANSP^S(n) \cap PPIN^S(n))$ and two profitability terms. The safety term ensures that it is safe to postpone distribution assignments to the entry/exit of some node. Hence, $PPOUT^S(n)$ denotes distribution assignments for which it is safe and profitable to sink them from node n in the direction of the control flow to successor nodes. $PPIN^S(n)$ denotes distribution assignments for which it is safe to place them at the entry of node n. Based on $PPIN^S$ and $PPOUT^S$, the sets $INSERT^S(n)$ and $INSSYNTH^S(m,n)$ denote distribution assignments which will be placed at the entry of node n or along the edge (m,n), respectively. Finally, $DEAD(n)$ describes distribution assignments which either have already been dead in node n before any transformation took place or have turned to dead due to the newly inserted distribution assignments.

The sinking of distribution assignments is guided by two *profitability terms* which shall prevent unnecessary code motions. The first term $PDDOUT$ requires that the distribution is partially dead, whereas the second term $\bigcup$ ensures that there is at least one path where the distribution is dead or can be moved further, respectively. The second profitability term is important particularly in the case of the edge placement approach, since otherwise sinking of distribution assignments could result in a serious *code proliferation* due to the insertion of synthetic nodes along all edges ending in successor nodes.

Similar to PRE systems, we must prove for PDED that the transformations are safe and profitable. It must be shown that for all nodes n, $DEAD(n)$ denote dead distribution assignments which can be deleted, and that on any program path the number of distribution assignments has not been increased. The proofs for the following theorems are to be found in Appendix A.

Theorem 1 (Dead Distribution Assignments). *After insertion of distribution assignments at the entry of all nodes n with $a \in INSERT^S(n)$ and in all synthetic nodes mn with $a \in INSSYNTH^S(m,n)$ respectively, the last occurrence of the distribution assignment is dead in each node i with $a \in POSTLOC(i) \cap PPOUT^S(i)$.*

Theorem 2 (Profitability). *The PDED algorithm does not increase the number of distribution assignments along any program path.*

6 Elimination of Partially Redundant Distribution Assignments

We have classical PRE systems [5] adapted for the elimination of partially redundant distribution assignments. The local sets denote upwards exposed distribution assignments ($ANTLOC(n)$), downwards exposed distribution assignments ($GEN^H(n)$), and transparent distribution assignments ($TRANSP^H(n)$). A comparison of the local sets of Table 1 and Table 2 shows that $POSTLOC \equiv GEN^H$, $GEN^S \equiv ANTLOC$, and $TRANSP^S \equiv TRANSP^H$. The global sets $AVIN(n)/AVOUT(n)$ and $PAVIN(n)/PAVOUT(n)$ denote distribution assignments which are available or partially available at entry/exit of node n, respectively.

$AVIN(n)$	$:=$	$\{(A,\delta) \in ADA : (A,\delta)$ available at entry of node n $\}$
$AVOUT(n)$	$:=$	$\{(A,\delta) \in ADA : (A,\delta)$ available at exit of node n $\}$
$PAVIN(n)$	$:=$	$\{(A,\delta) \in ADA : (A,\delta)$ partially available at entry of node n $\}$
$PAVOUT(n)$	$:=$	$\{(A,\delta) \in ADA : (A,\delta)$ partially available at exit of node n $\}$

Table 2: PRED Framework

Local Sets

$ANTLOC(n)$	$:=$	$\{$ $(A,\delta) \in ADA$: there is an occurrence of $dist(A) := \delta$ in n prior to any occurrence of A or definition of a variable used in δ $\}$
$GEN^H(n)$	$:=$	$\{$ $(A,\delta) \in ADA$: there is an occurrence of $dist(A) := \delta$ in n which is not followed by any other occurrence of A nor any definition of a variable used in δ $\}$
$TRANSP^H(n)$	$:=$	$\{$ $(A,\delta) \in ADA$: there is no occurrence of A nor definition of a variable used in δ in n $\}$

Data Flow Equations for Eliminating Partially Redundant Distribution Assignments

$PPOUT^H(n)$	$:=$	$\bigcap_{p \in succ(n)}(PPIN^H(p))$
$PPIN^H(n)$	$:=$	$(ANTLOC(n) \cup (TRANSP^H(n) \cap PPOUT^H(n))) \cap PAVIN(n) \cap (\bigcup_{p \in pred(n)}((PPIN^H(p) - ANTLOC(p)) \cup AVOUT(p)))$
$INSERT^H(n)$	$:=$	$(PPOUT^H(n) - AVOUT(n)) - (PPIN^H(n) \cap TRANSP^H(n))$
$INSSYNTH^H(m,n)$	$:=$	$PPIN^H(n) - (PPOUT^H(m) \cup AVOUT(m))$
$REDUND(n)$	$:=$	$ANTLOC(n) \cap PPIN^H(n)$

The data flow equations for the sets $AVIN/AVOUT$ are $AVIN(n) = \bigcap_{p \in pred(n)}(AVOUT(p))$ and $AVOUT(n) = GEN(n) \cup (TRANSP(n) \cup AVIN(n))$ whereby GEN and $TRANSP$ differ from GEN^H and $TRANSP^H$ in considering distribution assignments rather than array references for blocking flow information. Partial available distribution analysis differs from available distribution analysis only in the confluence operator which is changed from $\bigcap$ to $\bigcup$.

The $PPIN^H$ equation consists of the safety term $ANTLOC(n) \cup (TRANSP^H(n) \cap PPOUT^H(n))$ and two profitability terms. The safety term describes the property of anticipability which is identical to the *very busy* data flow problem [2] which assures that it is safe to hoist distribution assignments. The global sets $PPIN^H(n)/PPOUT^H(n)$ contain distribution assignments which may be placed at entry/exit of node n. The sets $INSERT^H(n)$ and $INSSYNTH^H(m,n)$ denote distribution assignments which will be placed at the exit of node n or along the edge (m,n), respectively. Finally, $REDUND(n)$ describes distribution assignments which either have already been redundant in node n or have become redundant by the newly inserted distribution assignments. The hoisting of distribution assignments is guided by the *profitability terms* $PAVIN$ and the $\bigcup$-term which ensures that there is at least one path where the distribution assignment is redundant or can be moved further, respectively.

7 Distribution Placement Framework

The distribution placement framework consists of eliminating partially dead distribution assignments followed by eliminating partially redundant distribution assignments. Thus, the resulting combined algorithm covers following optimizations: dead code elimination, suppression of partial dead code, redundancy elimination, suppression of partial redundancies, and elimination of loop-invariant distribution assignments by hoisting or sinking the assignments just before or just after the loop, respectively. The safety constraints which control the motion of distribution assignments have been chosen for both PDED and PRED such that it is guaranteed that the actual distribution for any array reference will be identical at runtime for the unoptimized and optimized version of the program. As discussed in Section 2 and shown by Lemma 1 and Lemma 2 (proof to be found in Appendix A), PDED must be performed prior to PRED in order to increase the effectiveness of distribution placement.

Algorithm 1 : Distribution Placement

A. Solve PDED system

(i) for each node n: compute local sets $POSTLOC(n)$, $DEF^S(n)$, and $TRANSP^S(n)$

(ii) resolution of the unidirectional system of equations yielding $DDIN(n)/DDOUT(n)$ and $PDDIN(n)/PDDOUT(n)$ for each node n

(iii) resolution of the bidirectional system of equations yielding $PPIN^S(n)/PPOUT^S(n)$ for each n

(iv) for each node n: for each $a \in INSERT^S(n)$ or for each $a \in INSSYNTH^S(m,n)$ insert distribution assignment a at the entry of node n or along the edge (m,n), respectively

(v) for each node n: for each $a \in DEAD(n)$ delete last occurrence of a in n

B. Solve PRED system

(i) for each node n: compute local sets $ANTLOC(n)$, $DEF^H(n)$, and $TRANSP^H(n)$

(ii) resolution of the unidirectional system of equations yielding $AVIN(n)/AVOUT(n)$ and $PAVIN(n)/PAVOUT(n)$ for each node n

(iii) resolution of the bidirectional system of equations yielding $PPIN^H(n)/PPOUT^H(n)$ for each n

(iv) for each node n: for each $a \in INSERT^H(n)$ or for each $a \in INSSYNTH^H(m,n)$ insert distribution assignment a at the exit of node n or along the edge (m,n), respectively

(v) for each node n: for each $a \in REDUND(n)$ delete first occurrence of a in n

---

Figure 2: Distribution Placement Framework

Lemma 1. *PRED cannot create (partially) dead distribution assignments.*

Lemma 2. *PDED may create (partially) redundant distribution assignments.*

The distribution placement framework is presented in Figure 2. The unidirectional system of equations can be solved easily using existing techniques (e.g. monotone data flow systems [2]). It has been shown that for reducible flow graphs the iterative approach [2] takes at most $d + 2$ iterations with d denoting the loop-connectedness of the graph [10]. Since in practice d is often 3 or less [14], the algorithm will terminate on the average after at most 5 iterations. Moreover, the unidirectional data flow problems needed for PDED and PRED are required for other optimizations as well. The resolution of the bidirectional data flow problems has been discussed in [5]. There it is claimed that splitting of bidirectional equations into a sequence of unidirectional ones reduces the analysis costs.

8 Application of the Algorithm

Figure 3 illustrates the application of the distribution placement algorithm to the flow graph of Figure 1. We focus our attention on the distribution assignments $dist(A) := \delta_1$ and $dist(A) := \delta_2$ of node 6. For the sake of clarity, we will use a_1 for (A, δ_1) and a_2 for (A, δ_2).

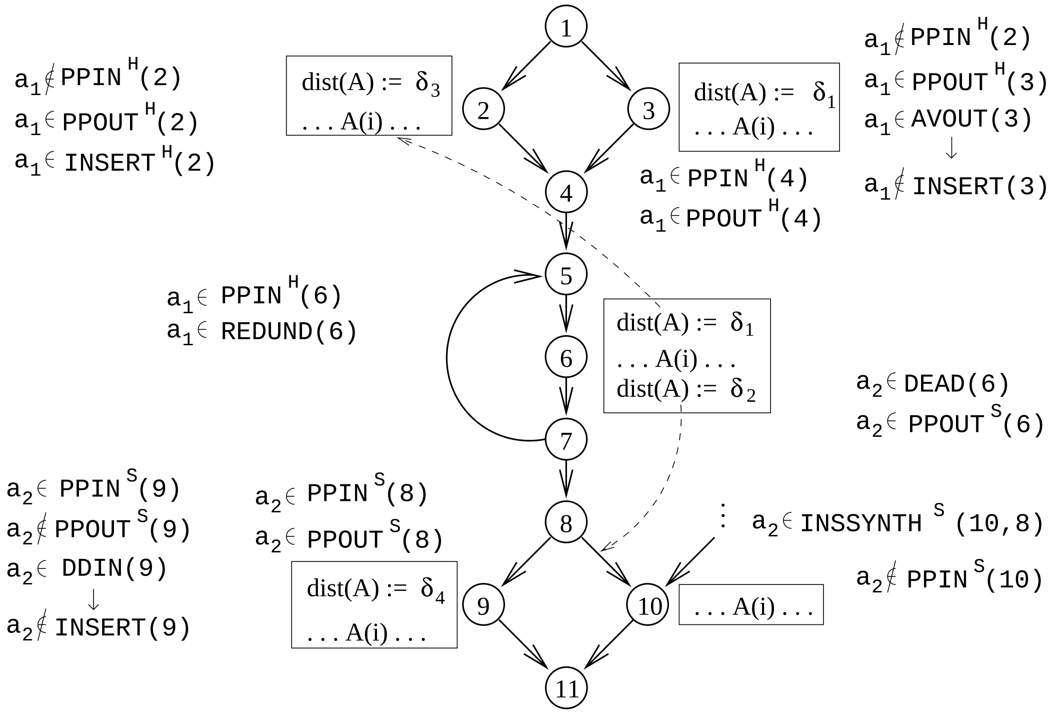


Figure 3: Algorithm applied to a simple flow graph.

First, PDED is applied. $a_2 \in POSTLOC(6) \cap PDDOUT(6)$ results in $a_2 \in PPOUT^S(6)$ which contributes to $a_2 \in PPIN^S(7)$, $a_2 \in PPOUT^S(7)$, $a_2 \in PPIN^S(8)$, and $a_2 \in PPOUT^S(8)$. Since $a_2 \notin PPIN^S(10)$, we can not sink the assignment to node 10 and eliminate in this way the deadness with respect to node 9. However, $a_2 \in INSSYNTH^S(8, 10)$ results in creating a synthetic node on edge $(8, 10)$ which contains the distribution assignment. $a_2 \in DEAD(6)$, $a_2 \notin INSERT^S(9)$, and $a_2 \in INSSYNTH^S(8, 10)$ has the effect of sinking the distribution assignment from node 7 to the synthetic node on the edge $(8, 10)$.

Now, PRED is applied. $a_1 \in ANTLOC(6) \cap PAVIN(6)$ causes a_1 to be in $PPIN^H(6)$ which results in $a_1 \in PPOUT^H(5)$, $a_1 \in PPIN^H(5)$, $a_1 \in PPOUT^H(4)$, and $a_1 \in PPIN^H(4)$. Since a_1 is in $PPOUT^H(2)$, we can place the assignment at node 2 and eliminate in this way the redundancy with respect to node 3. Although a_1 is in $ANTLOC(3)$, $a_1 \notin PAVIN(3)$ results in $a_1 \notin PPIN^H(3)$. $a_1 \in REDUND(6)$, $a_1 \notin INSERT^H(3)$, and $a_1 \in INSERT^H(2)$ has the effect of hoisting the distribution assignment from node 5 to the exit of node 2.

9 Experimental Results

To demonstrate the effectiveness of the presented optimization and to prove the expected performance results, we executed and measured on an Intel iPSC/860 hypercube and a Meiko CS-2 the unoptimized and optimized program versions of an Alternating-Direction-Implicit (ADI) integration kernel which is frequently used for solving partial differential equations. An ADI integration kernel consists of forward and backward sweeps along the rows followed by forward and backward sweeps along the columns. Hence, a row-wise data distribution requires communication for the sweeps along the columns but none along the rows – a column-wise data distribution requires communication vice versa. However, by choosing a row-wise distribution for the sweeps along the rows and a column-wise distribution for the column sweeps and redistributing the arrays between the two phases, no communication and synchronization is required within the sweeps at the expense of the remapping overhead.

```
      REAL, DIMENSION (N,N) :: x,a,b
!HPF$ DISTRIBUTE (BLOCK,BLOCK) :: x,a,b
...
DO iter=1,maxiter
   l1 :map2block-wise(x,a,b)
   CALL rows(x,a,b)
   l2 :map2block-block(x,a,b)
   l3 :map2column-wise(x,a,b)
   CALL columns(x,a,b)
   l4 :map2block-block(x,a,b)
END DO
...
```

```
      REAL, DIMENSION (N,N) :: x,a,b
!HPF$ DISTRIBUTE (BLOCK,BLOCK) :: x,a,b
...
DO iter=1,maxiter
   l1 :map2block-wise(x,a,b)
   CALL rows(x,a,b)
   l3 :map2column-wise(x,a,b)
   CALL columns(x,a,b)
END DO
l4 :map2block-block(x,a,b)
...
```

(a) Unoptimized and optimized version of ADI main loop.

```
SUBROUTINE rows(x,a,b)
REAL, DIMENSION (:,:) :: x,a,b
!HPF$ DISTRIBUTE (BLOCK,*) :: x,a,b
⋮ sweeps along rows
END
```

```
SUBROUTINE columns(x,a,b)
REAL, DIMENSION (:,:) :: x,a,b
!HPF$ DISTRIBUTE (*,BLOCK) :: x,a,b
⋮ sweeps along columns
END
```

(b) Subroutines performing the sweeps (for sweeps see [15]).

Figure 4: ADI integration kernel.

In Figure 4 the unoptimized and optimized program version of ADI is shown in HPF with the required remappings. The ADI code chosen here has been used by U. Kremer [15] to analyze the runtime behavior of such codes with different data distributions on an Intel iPSC/860 (for any details see [15]). Contrary to [15], we use subroutines to perform the sweeps along the rows and columns, respectively. Choosing subroutines instead of redistribute directives to perform distribution changes reflects also the current status of HPF-2, where dynamically distributed arrays have been moved from the base HPF language features to the Approved Extensions since an efficient implementation of dynamic data distributions is difficult. As a consequence, most commercial compilers do not support dynamic data distributions at the present at all or in a rather inefficient way.

As shown in Figure 4, in the unoptimized version of the ADI main loop the arrays x, a, and b are remapped four times just before and just after the subroutine calls *rows* and *columns* in order to establish the distributions requested by the dummy arguments and to restore the original ones, respectively. First, PDED is applied which recognizes the remapping at l_2 as dead. The remapping at l_4 is identified as dead within the loop but live with respect to the code following the loop. This

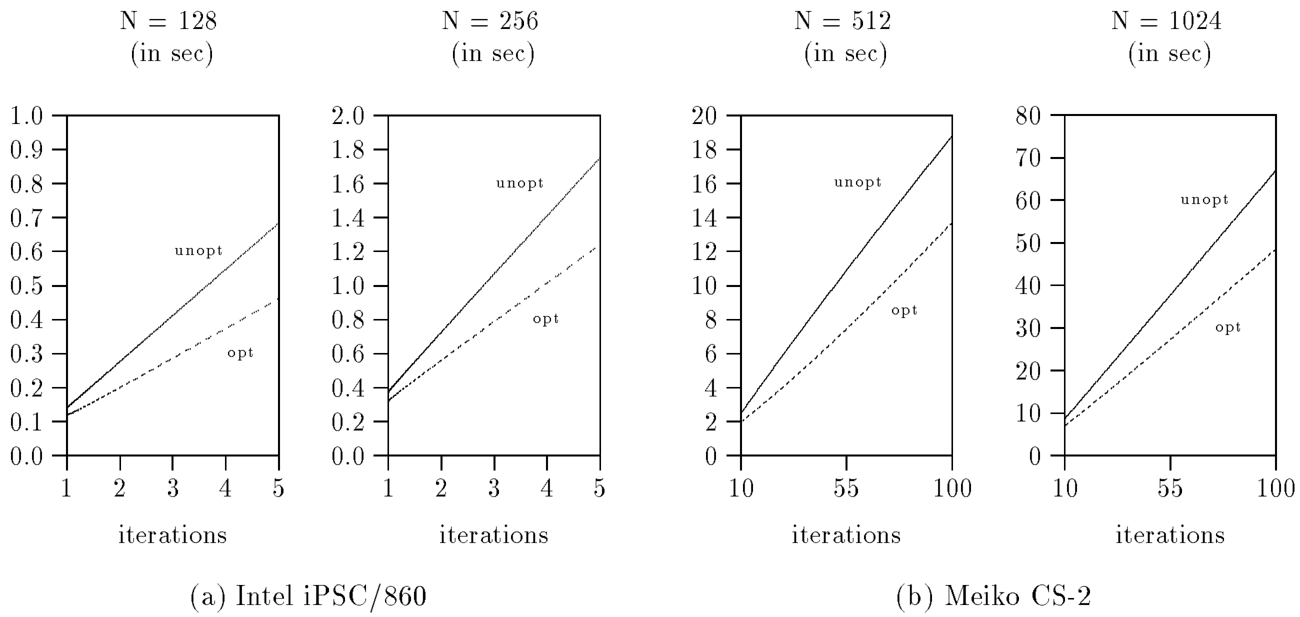


Figure 5: Measurements of ADI integration kernel for various problem sizes on different platforms.

partially deadness is eliminated by moving it just after the loop. The subsequently performed PRED fails to improve the code further since no partial redundancies are present. Hence, the 12 remappings required within one iteration for the arrays x, a, and b in the unoptimized version are reduced to 6 remappings within the loop and 3 remappings after the loop in the optimized program version.

Figure 5 presents the execution times of the ADI integration kernel for various problem sizes N measured on an Intel iPSC/860 hypercube and a Meiko CS-2. For both machines 16 processors and the NX message passing library have been used. The number of remappings which are performed in the optimized ADI version less than in the unoptimized version is given by r_k with k denoting the executed iterations as follows: $r_1 = 3$, $r_2 = 9$, $r_5 = 27$, $r_{10} = 57$, $r_{50} = 297$, and $r_{100} = 597$. The optimization results in a performance improvement which varies on the hypercube from 16% for 1 iteration to 48% for 5 iterations. On the Meiko we achieved a performance improvement from 24% for 10 iterations to 38% for 100 iterations.

10 Related Work

Morel and Renvoise [16] successfully introduced bidirectional data flows for suppressing partially redundant computations. Dhamdhere et al. [5] improved the Morel-Renvoise approach considerably by integrating the concept of edge placement and by reducing the resolution costs of bidirectional data flows by splitting them into a series of unidirectional data flows which are solved instead.

Knoop, Rüthing, and Steffen [11] proposed for suppression of partially redundant computations a lazy code motion strategy which places computations as early as necessary but as late as possible. Instead of using bidirectional data flows, it is realized by a sequence of unidirectional equations. Elimination of partially dead as well as partially redundant assignments is discussed in [12] and [13], respectively.

Partial redundancy elimination techniques have been applied by several researchers for optimizing communication in the field of high-performance compilers. Gupta et al. [8] applied PRE techniques to available section descriptors in order to reduce communication costs by capturing optimizations like vectorizing communication, removing redundant communication, or latency hiding. Hanxleden et al. [17] described a data flow framework which is based on a general producer-consumer concept and which is applicable to a broad class of placement problems whereby some limitations of classical PRE systems have been overcome. Agrawal et al. [1] addressed the problem of performing partial redundancy elimination interprocedurally which was motivated by communication schedule optimizations.

Only little work has been done on compilation techniques for handling dynamic redistributions efficiently. Hall et al. [9] stressed the importance of optimizing dynamically redistributions also across procedure boundaries. But instead of a uniform framework, individual algorithms are described for the various interprocedural optimizations. In [4] dynamic data distributions are discussed and a basic implementation is presented.

11 Conclusions

An optimization framework has been presented which reduces the number of executed distribution assignments. Instead of implementing individual optimizations, the distribution assignment placement framework covers various transformations in a uniform manner by solving more general problems, namely the suppression of partially redundant and partially dead distribution assignments, which results also in more powerful optimizations. The safety constraints which control code motion have been chosen for both PDED and PRED such that it is guaranteed that the actual distribution for any array reference is identical at runtime for the optimized and unoptimized version of the program. The effectiveness of the algorithm is increased by performing PDED prior to PRED. Edge splitting, a transformation which inserts new nodes on any edge running from a branch node (more than one successor) to a join node (more than one predecessor), which can make the flow graph unnecessarily large (e.g. a node is inserted for each if-statement without an else-branch), is not required prior to our analysis. Both PDED and PRED support insertion of synthetic nodes whenever beneficial.

Instead of analyzing the program at source level, we have introduced distribution assignments which offers two benefits. On the one hand, an abstraction from high-level language constructs is possible which permits a generic description of the optimization phase. On the other hand, we are able to handle dynamic realignments. The framework is currently being implemented within the VFCS system [3].

Future research will focus on handling second order effects and extending the framework to perform distribution assignment motion interprocedurally. Currently, alignments which cannot be removed or variables used in distribution specifications may introduce second order effects which may prevent some profitable transformations. Furthermore, splitting distribution assignments into sending and receiving components and hiding communication overhead will be investigated.

References

[1] G. Agrawal, J. Saltz, and R. Das. Interprocedural partial redundancy elimination and its application to distributed memory compilation. In *Proc. of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI'95)*, pages 258–269, La Jolla, CA, June 1995.

[2] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading,MA, 1988.

[3] S. Benkner, S. Andel, R. Blasko, P. Brezany, A. Celic, B.M. Chapman, M. Egg, T. Fahringer, J. Hulman, E. Kelc, E. Mehofer, H. Moritsch, M. Paul, K. Sanjari, V. Sipkova, B. Velkov, B. Wender, and H.P. Zima. *Vienna Fortran Compilation System - Version 1.2 - User's Guide.* Vienna, Austria, February 1996.

[4] B. Chapman, P. Mehrotra, H. Moritsch, and H. Zima. Dynamic data distribution in Vienna Fortran. In *Proc. Supercomputing '93*, Portland, Oregon, November 1993.

[5] D.M. Dhamdhere and H. Patil. An elimination algorithm for bidirectional data flow problems using edge placement. *ACM TOPLAS*, 15(2):312–336, April 1993.

[6] High Performance Fortran Forum. High Performance Fortran language specification version 2.0. Technical report, Rice University, Houston,TX, October 1996. Also available via HPFF home page: http://www.crpc.rice.edu/HPFF.

[7] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M.-Y. Wu. FORTRAN D language specification. Technical report, Rice University, Houston,TX, January 1992.

[8] M. Gupta, E. Schonberg, and H. Srinivasan. A unified data-flow framework for optimizing communication. In *Proc. of the 7th Workshop on Languages and Compilers for Parallel Computing*, Ithaca, NY, August 1994.

[9] M. W. Hall, S. Hirandani, K. Kennedy, and C.-W. Tseng. Interprocedural compilation of Fortran D for MIMD distributed-memory machines. In *Proc. of Supercomputing '92*, Minneapolis, November 1992.

[10] J.B. Kam and J.D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, January 1976.

[11] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. In *Proc. of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation (PLDI'92)*, pages 224–234, San Francisco,CA, 1992.

[12] J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. In *Proc. of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI'94)*, pages 147–158, Orlando, Florida, June 1994.

[13] J. Knoop, O. Rüthing, and B. Steffen. The power of assignment motion. In *Proc. of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI'95)*, pages 233–245, La Jolla,CA, June 1995.

[14] D.E. Knuth. An empirical study of FORTRAN programs. *Software Practice and Experience*, 1(2):105–134, April 1971.

[15] U. Kremer. *Automatic data layout for distributed memory machines.* PhD thesis, Rice University, Houston,TX, October 1995.

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

[17] R. von Hanxleden and K. Kennedy. GIVE-N-TAKE - A Balanced Code Placement Framework. In *Proc. of the ACM SIGPLAN '94 Conference on Program Language Design and Implementation*, Orlando, June 1994.

[18] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran - A language specification version 1.1. Technical Report ACPC/TR 92-4, Austrian Center for Parallel Computation, March 1992.

Appendix A Proofs

Similar to PRE systems, we must prove for PDED that the transformations are safe and profitable. It must be shown that for all nodes n, $POSTLOC(n) \cap PPOUT^S(n)$ denote dead distribution assignments which can be deleted, and that on any program path the number of distribution assignments has not been increased. Moreover, it is shown that it is profitable to perform PDED prior to PRED.

Lemma 3. *After insertion of distribution assignments at the entry of all nodes n with* $a \in INSERT^S(n)$ *and in all synthetic nodes mn with* $a \in INSSYNTH^S(m,n)$ *respectively, the updated set* $DDOUT(k)$ *for any node k with* $a \in PPOUT^S(k)$*, denoted by* $DDOUT'(k)$*, will contain a.*

Proof. Assume that $a \in PPOUT^S(k)$ and $a \notin DDOUT'(k)$. Then there must exist a node i such that $a \notin DDIN(i) \cup INSERT^S(i) \cup TRANSP^S(i)$, and the path $]k,i[$ is transparent with $a \notin INSERT^S(u)$, $u \in]k,i[$, and $a \notin INSSYNTH^S(v',v)$, $v \in]k,i]$, respectively. The first condition expresses that the distribution assignment is live and the second condition assures that no new distribution assignment has been inserted which turned the distribution assignment to dead. Then, $a \notin DDIN'(i)$ is propagated by the confluence operator from i to k till $a \notin DDOUT'(k)$.

Since $a \notin INSERT^S(i) \cup DDIN(i) \cup TRANSP^S(i)$, we get $a \notin PPIN^S(i)$ by the $INSERT$ equation. Let j be the predecessor of i in the path $]k,i[$. Then, $a \notin INSSYNTH^S(j,i)$, $a \notin PPIN^S(i)$ and $a \notin DDIN(i)$ implies $a \notin PPOUT^S(j)$ by the $INSSYNTH$ equation. If

a. $j = k$ we have a contradiction on $PPOUT^S(k)$ which concludes the proof.

b. Otherwise $a \notin INSERT^S(j)$ together with $a \notin DDIN(j)$ [1] and $a \notin PPOUT^S(j)$ implies $a \notin PPIN^S(j)$ by the $INSERT$ equation. Now, the same considerations can be applied for the predecessor of node of j and so on until we reach node k and branch a) matches. ∎

Theorem 1. Dead Distribution Assignments (see Section 5).

Proof. Obviously, $a \in POSTLOC(i) \cap PPOUT^S(i)$ implies $a \in PPOUT^S(i)$. By Lemma 3, $a \in DDOUT'(i)$. Since $POSTLOC(i)$ can not be changed by insertions, $a \in POSTLOC(i) \cap DDOUT'(i)$ which indicates dead code. ∎

Lemma 4. *Every program path ending at the entry of node i with* $a \in INSERT^S(i)$ *or* $a \in INSSYNTH^S(j,i)$*, contains a dead distribution assignment.*

Proof. On the one hand, $a \in INSERT^S(i)$ implies $a \in PPIN^S(i)$ and $a \in POSTOUT(j)$, $j \in pred(i)$. On the other hand, $a \in INSSYNTH^S(j,i)$ implies $a \in PPOUT^S(j)$ and $a \in POSTOUT(j)$. Hence, there exists an assignment on every path ending at node i. We have to show now that these assignments are dead.

Assume a node which contains an assignment which is not dead. By Theorem 1, $a \notin POSTLOC(k) \cap PPOUT^S(k)$ with path $]k,i[$ being transparent. Since $a \in POSTLOC(k)$, $a \notin PPOUT^S(k)$ leads to $a \notin PPIN^S(l)$ for $l \in succ(k)$ and $l \in]k,i]$. If

a. $l = i$ we have a contradiction on $PPIN^S(i)$ or $PPOUT^S(j)$ respectively which concludes the proof.

b. Otherwise, $a \notin PPIN^S(l)$, $a \in TRANSP^S(l)$, and $a \notin POSTLOC(l)$ leads to $a \notin PPOUT^S(l)$. Now, the same considerations can be applied for the successor of node l and so on until we reach node i and branch a) matches. ∎

[1] $a \notin DDIN(i)$ implies $a \notin DDOUT(j)$ which leads to $a \notin DDIN(j)$ since $]k,i[$ is transparent.

Lemma 5. *Any path starting from the exit of a node k with $a \in DEAD(k)$ and ending in a node i such that $a \in INSERT^S(i)$ or $a \in INSSYNTH^S(j,i)$, can not contain another node v with either $a \in INSERT^S(v)$ or $a \in INSSYNTH^S(v,u)$.*

Proof. By Lemma 4, $a \in POSTLOC(k) \cap PPOUT^S(k)$ with path $]k,i[$ being transparent. Assume that there exists an $u \in]k,i[$ with $a \in INSERT^S(u)$ or $a \in INSSYNTH^S(u',u)$. From $a \in INSERT^S(u)$ we get $a \notin PPOUT^S(u)$. Since $a \in INSSYNTH^S(u',u)$ implies $a \notin PPIN^S(u)$ and since $a \notin POSTLOC(u)$, we get $a \notin PPOUT^S(u)$.

As we have seen in the proof of Lemma 4, $a \notin PPOUT^S(u)$ propagates through $]u,i[$ until $a \notin PPOUT^S(j)$, $j \in pred(i)$, and finally $a \notin PPIN^S(i)$. However, $a \notin INSERT^S(i)$ or $a \notin INSSYNTH^S(j,i)$ is a contradiction which concludes the proof. ∎

Theorem 2. Profitability (see Section 5).

Proof. Lemma 4 and Lemma 5 imply Theorem 2. ∎

Lemma 1 (see Section 7).

Proof. Hoisting of a distribution assignment of the form $dist(A) := \delta$ is blocked by any reference to array A. Thus, hoisting cannot affect (partial) dead distribution analysis. ∎

Lemma 2 (see Section 7).

Proof. Consider Figure 1. Sinking of the second distribution assignment from node 6 to the edge $(8,10)$ turned the first distribution assignment to be redundant within the loop. ∎