Interprocedural Distribution Assignment Placement: More than just Enhancing Intraprocedural Placing Techniques *

Jens Knoop

Fakultät für Mathematik und Informatik
Universität Passau, Germany
(email: knoop@fmi.uni-passau.de)

Eduard Mehofer

Inst. f. Softwaretechnik u. Parallele
Systeme, Universität Wien, Austria
(email: mehofer@par.univie.ac.at)

Abstract

Avoiding unnecessary remappings at run-time by means of a strategic placement of distribution assignments (DAP) *is a major means for improving the run-time efficiency of data-parallel programs on distributed-memory architectures. Recently, we presented a novel and aggressive intraprocedural algorithm achieving this by eliminating partially redundant and partially dead distribution assignments. Here, we show how to enhance this approach interprocedurally. Surprisingly at first sight, it turns out that a straightforward adaption of the intraprocedural approach fails because central properties being valid for the intraprocedural case do not carry over to the interprocedural one revealing severe anomalies. After discussing the essential differences and analogies of* DAP *in the intraprocedural and interprocedural case, we show how to overcome these anomalies in order to arrive at a powerful and flexible approach for interprocedural* DAP (IDAP)*. As in the intraprocedural case we get a hierarchy of* IDAP*-algorithms of varying power and efficiency supporting user-customized solutions. First practical experiences underline its importance and effectivity.*

Keywords: Data-parallel languages, High Performance Fortran (HPF), (dynamic) data redistribution, (interprocedural) data-flow analysis, optimization, (interprocedural) partially dead/redundant assignment elimination.

1 Motivation

Data-parallel languages like High-Performance Fortran (HPF) [4], Fortran D [5], and Vienna Fortran [18] allow to adapt the mapping of arrays to varying computational kernels or varying processor workloads at subprogram boundaries and with redistribute/realign directives. In order to model the different sources for data remappings uniformly, we introduce distribution assignments which establish an association between an array and a distribution. They are generated by the compiler whenever a remapping of an array takes place [14]. Distribution assignments can be very expensive because usually big amounts of data have to be moved. Avoiding them where possible is of crucial importance for achieving high performance on distributed memory architectures.

In [10] we recently presented a novel and aggressive approach for intraprocedural *distribution assignment placement* (DAP) which reduces the number of remappings by repeatedly applying *partially dead code elimination* (PDCE) and *partially redundant assignment elimination* (PRAE). This is necessary in order to fully exploit the *second-order effects* (cf. [17]) of PDCE and PRAE. However, performing DAP for every procedure separately can prevent beneficial optimizations, a fact, which was also pointed out in [7] showing that *interprocedural* compilation is needed for generating highly efficient code.

This is illustrated in Figure 2. It demonstrates the power of our IDAP-approach, which results from properly combining interprocedural PDCE and PRAE (for short: IPDCE and IPRAE) by means of a typical example showing the benefits of moving distribution assignments across subprogram boundaries. The dummy array of subroutine $F2$ is prescriptively mapped in a column-wise manner onto the processors. According to our basic code generation strategy, the caller establishes the mappings requested by the callee [14], i.e., two distribution assignments are inserted in $F1$ just before and just after the call of $F2$ in order to establish the column-wise mapping and to restore the original one. The point of this example is that the

*In *Proc. of the IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT'97)*, pages 26 – 37, San Francisco, CA, November 1997.

second remapping is *partially dead* because it is only used along program continuations satisfying the condition of the branch statement in the main procedure P. Interprocedural PDCE succeeds in moving this remapping from $F1$ to the main program and, subsequently, out of the loop to its only use site inside the branch statement. As a side-effect, the elimination of this partially dead distribution assignment turned the other one to be *partially redundant* within the loop. Thus, as a second-order effect, interprocedural PRAE hoists this remapping from $F1$ to the caller and places it just before the loop. Together, this reduces the number of distribution assignments executed at run-time dramatically. The optimized program is free of any partially dead and partially redundant distribution assignment. Moreover, this example illustrates another important feature: the mapping which has to be restored after a subroutine call is usually run-time dependent as it can be called in different contexts. As PDCE places assignments in more specific contexts, it eases as a side-effect the problem of determining the restoring mapping at compile-time, which is demonstrated in Figure 2 where *restore* has been replaced by $(block, *)$ after assignment sinking.

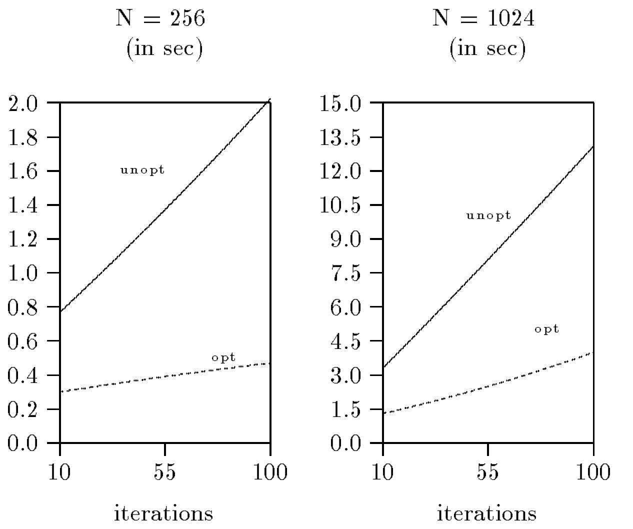

Figure 1: Measured run-times for the program of Fig. 2 on a Meiko CS-2 with 16 processors.

In addition to illustrating the power of our IDAP-approach, Figure 2 also reveals the conceptual analogy of interprocedural DAP to its intraprocedural counterpart (cf. [10]). This suggests that the concepts underlying intraprocedural DAP carry over straightforward to the interprocedural setting. However, as we are going to demonstrate, there are essential differences between the intraprocedural and interprocedural setting. As a consequence, successful IDAP requires more than just enhancing intraprocedural placing techniques. Most important, the optimality results applying to intraprocedural PDCE and PRAE (cf. [11, 12]) are impossible in general in the interprocedural setting. Moreover, the strategy of moving assignments "as-far-as-possible", which leads to optimal results in the intraprocedural setting, can exhibit severe anomalies in the interprocedural setting. However, we will show how to overcome these anomalies. As in the intraprocedural setting we arrive at a hierarchy of IDAP-algorithms of varying power and efficiency fitting a user's individual needs.

Practical experiences. In order to demonstrate the effectivity of IDAP, we measured both the original and the optimized version of the code shown in Figure 2. Subroutine $F2$ sweeps over the columns and calculates a new value for $A(i, j)$ by looking at the neighboring elements $A(i-1, j)$ and $A(i+1, j)$. Choosing a column-wise mapping, no communication and synchronization is required within the sweeps. It is assumed that in the main program array A is distributed by $(block, *)$. For both versions we measured the execution time of the do-loop of the main program with subroutine $F1$ containing only function call $F2$. For the optimized version, we added also the time required to restore the mapping of array A, i.e., we measured in our test the worst case behavior assuming that the then-branch is always executed. The results are shown in Figure 1. We ran our tests on 16 processors of a Meiko CS-2 distributed memory architecture with the MPI message passing library. The performance measurements of Figure 1 demonstrate the importance of this kind of optimization for varying problem sizes and iterations. The dramatic improvement achieved is not surprising in the light of average remapping times (cf. [16]).

Related work. Remapping analysis has been addressed by several researchers. Hall et al. [7] stressed the importance of interprocedural compilation and presented techniques for hoisting remappings out of loops and eliminating dead remappings. Contrary to our approach, the more general problem of eliminating partially dead and partially redundant remappings is not considered. Coelho et al. [3] describe an optimization which reduces the communication amount by removing useless remappings and taking advantage of replications to shorten individual remappings. The elimination of useless remappings is based on a remapping graph which is presented at an intraprocedural level. Optimal in their sense means that for a given remapping, a minimal number of messages is sent over the network. However, they do not use code

(a) Basic distribution assignment generation.

```
! main program
program P
real A(N,N)
···
! assumed: A ≡ (block,*)
do iter=1,MAXITER
  call F1
end do
···
if (cond) then
  ··· A(i) ···
end if
end
```

```
! internal subprogram
subroutine F1
··· no remap, no A
dist(A):=(*,block)
call F2(A,N)
dist(A):=restore
··· no remap, no A
end
```

```
! internal subprogram
subroutine F2(X,N)
real X(N,N)
!HPF$ distribute X(*,block)
do j=1,N
   do i=2,N-1
      X(i,j)=X(i,j)/2.0 + (X(i-1,j)+
+               X(i+1,j))/4.0
   end do
end do
end
```

(b) Result of interprocedural DAP based on interprocedural PDCE and PRAE.

```
! main program
program P
real A(N,N)
···
! assumed: A ≡ (block,*)
dist(A):=(*,block)
do iter=1,MAXITER
  call F1
end do
···
if (cond) then
  dist(A):=(block,*)
  ··· A(i) ···
end if
end
```

```
! internal subprogram
subroutine F1
··· no remap, no A
call F2(A,N)
··· no remap, no A
end
```

```
! internal subprogram
subroutine F2(X,N)
real X(N,N)
!HPF$ distribute X(*,block)
do j=1,N
   do i=2,N-1
      X(i,j)=X(i,j)/2.0 + (X(i-1,j)+
+               X(i+1,j))/4.0
   end do
end do
end
```

Figure 2: Illustrating the power of interprocedural distribution assignment placement.

motion in order to reduce the overall number of executed data remappings, which is the central topic of this article. Palermo et al. [15] present several remapping analyses and transformations including reaching distribution analysis and making implicit remappings at subprogram boundaries explicit. Motion of data remappings, however, is not considered.

Several researchers applied techniques based on *partially redundant expression elimination* (PREE) for communication optimization. Gupta et al. [6], e.g., applied PREE-techniques to available section descriptors in order to perform a variety of communication optimizations. Agrawal et al. [1] focused on the problem of interprocedural placement of communication schedules. Kennedy et al. [9] developed a communication placement framework based on PREE-techniques which supports constraint-based latency hiding. None of the above approaches combines code hoisting (PRAE) with code sinking (PDCE) techniques.

Structure of the article. After presenting our preliminaries in Section 2, we discuss the essential differences and analogies of intraprocedural and interprocedural DAP in Section 3. Central is then Section 4, where we introduce a natural and sufficient constraint $\mathcal{C}$ for avoiding anomalies during interprocedural assignment motion (Section 4.1), present our hierarchy of IDAP-algorithms (Section 4.2), and discuss their algorithmic implementations (Section 4.3). Finally, we draw our conclusions in Section 5.

2 Preliminaries

We consider programs Π as systems $\langle \pi_1, \ldots, \pi_k \rangle$ of (mutually recursive) procedure (subprogram) definitions, where each $\pi \in \Pi$ has a list of formal value and reference parameters, and a list of local variables. The procedure π_1 is assumed to denote the *main procedure* of Π and therefore cannot be called. π_2 up to π_k are the *procedure* (*subprogram*) *declarations* of Π. For simplicity we assume that different declaring occurrences of names use different identifiers and that there is no (static) subprogram nesting except that π_1 encloses π_2 up to π_k. The variables of the main program are thus global variables of the subprograms, and can be accessed by them. Procedures are represented by means of directed edge-labeled flow graphs $G = (N, E, \mathbf{s}, \mathbf{e})$ with node set N, edge set E, and a unique *start node* $\mathbf{s}$ and *end node* $\mathbf{e}$, which are assumed to have no incoming and outgoing edges, respectively. Edges $e \in E$ represent (ordinary) *assignments*, *distribution assignments* of the form $dist(A)$:= *"HPF-mapping"*, subprogram calls, and output operations of the form $out(t)$ forcing all operands of term t to be alive. Distribution assignments are generated by the compiler whenever a remapping takes place (cf. [14]). They uniformly denote distribution changes occurring throughout the program caused by subprogram calls or by redistribute/realign directives. Unlabeled edges are assumed to represent "skip". Programs Π are then represented as systems $S =_{df} \langle G_1, \ldots, G_k \rangle$ of flow graphs with disjoint sets of nodes N_i and edges E_i. Finally, $E_{call} \subseteq E =_{df} \bigcup \{E_i \mid i \in \{1, \ldots, k\}\}$ denotes the set of all edges of S representing a subprogram call.

3 Intra- vs. interprocedural DAP

3.1 Optimality of PDCE and PRAE

As recalled above the effect of DAP relies on the combined effects of PDCE and PRAE (cf. [10]). Conceptually, also PDCE and PRAE are composed of two elementary transformations each: admissible *assignment sinkings* (AS) and *dead code eliminations* (DCE), and admissible *assignment hoistings* (AH) and *redundant assignment eliminations* (RAE), respectively.

Intraprocedurally, both PDCE and PRAE can be organized *optimally*, i.e., for every argument program Π there exists a program Π_{opt}, which is "globally best". Π_{opt} can effectively be constructed, and it is *better* than any other program in the set $\mathcal{G}$ of programs derivable from Π by sequences of admissible assignment sinkings and dead code eliminations, or assignment hoistings and redundant assignment eliminations, respectively. Note, a program $G' \in \mathcal{G}$ is better than a program $G'' \in \mathcal{G}$ if and only if the number of assignments executed on each path in G' is less or equal to that in G''. Assignments remaining in Π_{opt} after PDCE or PRAE, respectively, cannot be eliminated further without changing the branching structure or the semantics of the program, or without impairing some program executions (cf. [11, 12]).

Interprocedurally, a globally best program exists neither for PDCE nor for PRAE in general. In general, there are *incomparable minima* only as it is demonstrated for PRAE by the program Π of Figure 3. Note that there are only two programs being "significantly" different and "better" than Π, which are displayed in Figures 4 and 5. Both Π' and Π'' are "better" than Π, but themselves are incomparable with respect to this relation. This can easily be checked by means of the table below summarizing the number of assignments executed in the three programs.

Program	paths via π_2	
	"left" path	"right" path
Π	2	1
Π'	1	1
Π''	2	1
Program	paths via π_4	
	"left" path	"right" path
Π	3	2
Π'	2	2
Π''	2	1

3.2 Moving "as-far-as-possible"

Intraprocedurally, the strategy underlying the sinking (hoisting) of assignments is moving them *as-far-as-possible* because this maximizes the potential of dead (redundant) code without affecting the program semantics or performance (in terms of statements executed). In essence, for sinkings (hoistings) this means moving assignments to program points having a successor (predecessor), where placing an occurrence would affect the program semantics or performance. As recalled above, intraprocedurally this strategy leads to optimal results.

Interprocedurally, the very same strategy can exhibit severe anomalies. This is demonstrated by the program of Figure 6. Note that all distribution assignments are totally live. Thus, IPDCE does not have any effect. Note further that the distribution assignment in π_4 is *necessary*, i.e., it is not partially redundant. Considering the occurrences of $dist(A) := cyclic(k)$, the application of the as-far-as-possible hoisting strategy followed by an elimination of redundant assignments results in the program of Figure 7.[1] Since π_3 is also called by subprogram π_2, $dist(A) := cyclic(k)$ is blocked in π_4 at the second call of π_3 because moving it inside π_3 would affect the performance of program

[1] In Figure 6, the dashed lines indicate the distribution assignments which justify the new locations marked by crosses.

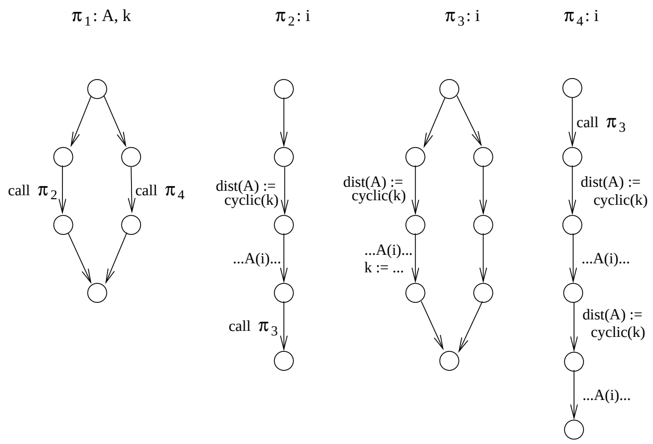

Figure 3: No optimality in general: The original program Π.

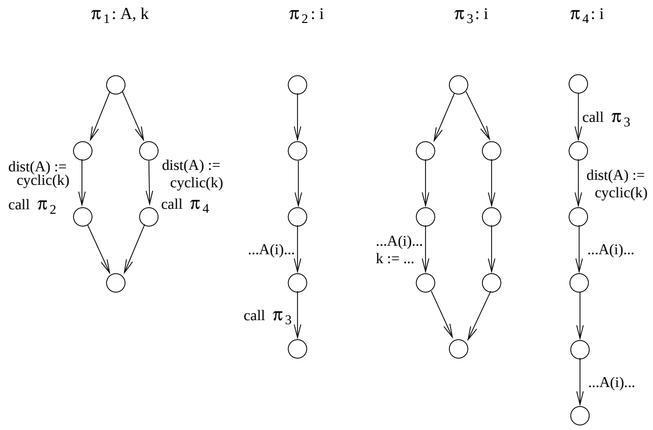

Figure 4: No optimality in general: A first minimal program Π'.

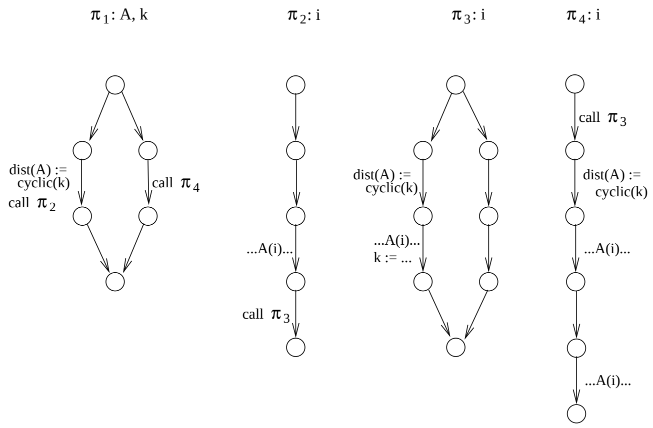

Figure 5: No optimality in general: A second mininal program Π''.

executions calling π_3 via π_2. On the other hand, the occurrence of $dist(A) := cyclic(k)$ just after the second call of π_3 in π_4 together with the occurrence in π_3, allows us to hoist the distribution assignment through π_3, and to place it in front of the second call of π_3 in π_4. Precisely the same applies to the first call of π_3 in π_4 and, finally, $dist(A) := cyclic(k)$ is placed in π_1, where hoisting stops. Unfortunately, this transformation increased the number of executed distribution assignments for subprogram π_4 (in fact, the number of distribution assignments can be increased arbitrarily by adding new subprogram calls of π_3 in π_4).

We remark that the anomalies above only show up for "global" assignment patterns, i.e., for patterns, where the scope of all variables is global. For assignment patterns involving local variables "mixed" situations as above, where movability of an assignment across a procedure call is justified in part by assignment occurrences inside the caller and occurrences inside the callee cannot occur as they refer to different variable incarnations. This will be important for the construction of our IDAP-algorithms (cf. Section 4.1).

3.3 Uniform handling of assignment patterns

In general, both distribution and ordinary assignments must be taken into account in order to exploit the full power of DAP because ordinary assignments can block the motion of distribution assignments preventing thus their elimination (cf. [10]). *Intraprocedurally*, all assignment patterns (ordinary and distribution assignments) can be handled uniformly, i.e., the admissibility of a specific sinking (hoisting) strategy does not depend on the assignment pattern under consideration.

Interprocedurally, "recursive" assignment patterns, i.e., assignment patterns, whose left-hand side variable occurs in their right-hand side term like in $u := u + 1$, and "non-recursive" ones behave differently. This is illustrated in the example of Figure 8. It shows a situation, which is completely symmetric for the assignment patterns $u := u + 1$ and $dist(A) := cyclic(i)$. Whereas, however, the transformation displayed in Figure 9 is perfectly fine for the non-recursive pattern $dist(A) := cyclic(i)$, the same transformation for the pattern $u := u+1$ affects the program semantics: both assignments to i occurring in the program of Figure 9 lead along program executions via π_4 following the left branch of π_3 to different values than in the program of Figure 8.[2]

[2] Only ordinary assignment patterns can be "recursive" – "recursive" distribution patterns are not possible in HPF.

4 Interprocedural distribution assignment placement

In this section we present our interprocedural approach to DAP. Because of the missing optimality of IPDCE and IPRAE in general, which also excludes global optimality of IDAP in general, and because of the anomalies, which may result from a naive adaption of the as-far-as-possible motion strategy for global assignment patterns, we focus on the development of a *practical* and *powerful* approach for IDAP, which in the absence of procedures reduces to its intraprocedural counterpart, avoids all the anomalies of a naive extension of the underlying intraprocedural approach, and simultaneously preserves its major benefits, in particular, (1) *uniformity*, i.e., the capability of handling all assignment patterns in the same fashion, and (2) *flexibility* allowing user-customized solutions by easily trading efficiency against power and vice versa.

Intuitively, this is achieved by imposing a natural constraint $\mathcal{C}$ on the movability of global assignment patterns across procedure calls, which guarantees that a callee behaves "similar" for all its call sites with respect to the pattern under consideration (cf. Section 4.1). It is important that this constraint can be encapsulated into the generic algorithm for interprocedural data-flow analysis (IDFA) of [13] underlying our approach, and thus, it does not need to be explicitly dealt with on the application level of the framework when specifying the interprocedural versions of the hoistability and sinkability analyses. As in the intraprocedural case, we arrive at a hierarchy of algorithms for IDAP of varying power and efficiency fitting a user's requirements. Like their intraprocedural counterparts the algorithms are open for refinements in order to further enhance their *power*, e.g., by considering partially faint code elimination instead of partially dead code elimination, and *efficiency*, e.g., by *decoupling* of (distribution) patterns and *masking* of distribution assignments (cf. [10]).

4.1 Avoiding hoisting and sinking anomalies: The motion constraint $\mathcal{C}$

As illustrated in Section 3, moving global assignment patterns across subprogram calls requires special care in order to (1) avoid anomalies, and (2) to enable a uniform treatment of all assignment patterns. In our approach, this is accomplished by imposing a natural and sufficient constraint $\mathcal{C}$ on moving global assignment patterns across subprogram calls. Intuitively, $\mathcal{C}$ guarantees that occurrences of a global pattern are moved across a subprogram call only, if

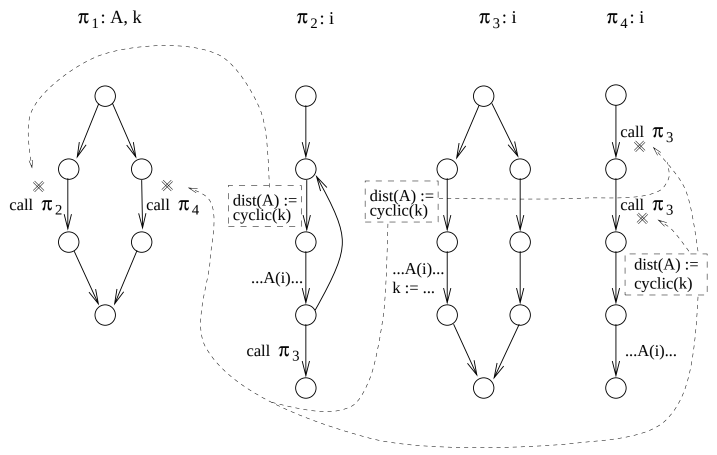

Figure 6: Moving assignments "as-far-as-possible" can lead to anomalies.

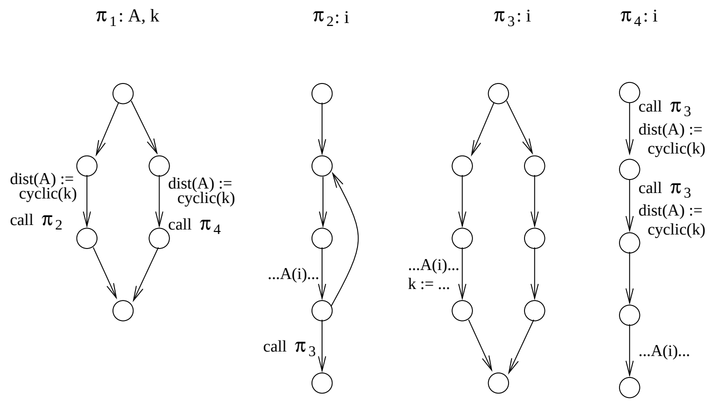

Figure 7: Making the transformation of Figure 6 explicit.

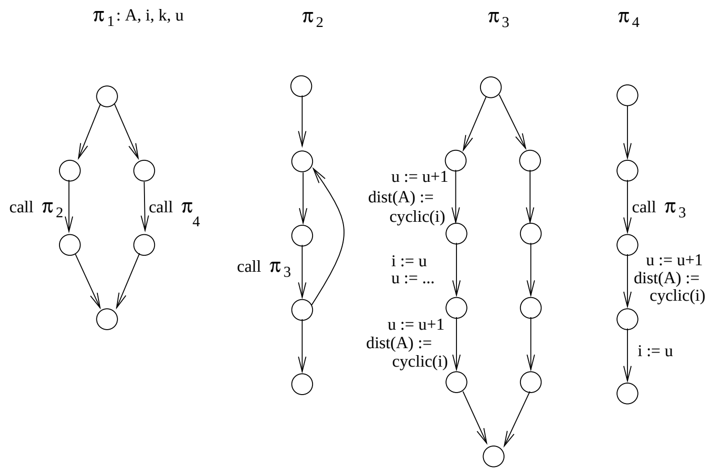

Figure 8: Illustrating the impact of "recursivity" of assignment patterns.

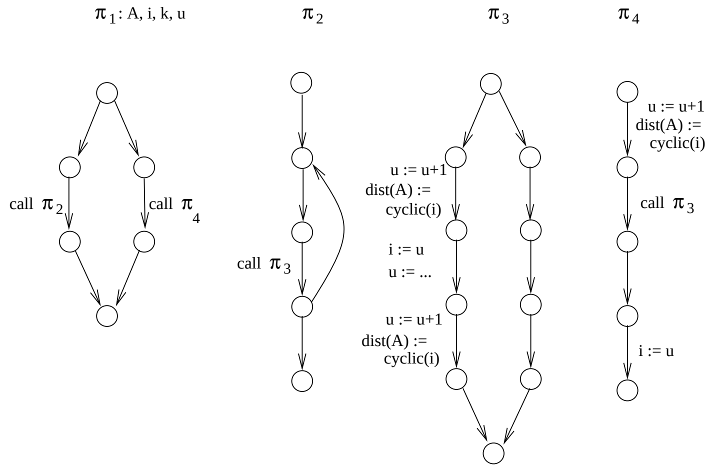

Figure 9: Treating "recursive" and "non-recursive" patterns analogously can lead to anomalies.

- *Uniformity*: the start and end node of the callee satisfy the motion predicate (hoisting or sinking) under consideration, or if
- *Independence*: the callee and all subprograms (indirectly) callable by it are free of statements blocking the pattern under consideration.

Intuitively, the first part of constraint $\mathcal{C}$ imposes a *uniformity* condition on all call sites of a callee. It excludes situations like in Figure 6, where from the point of view of some call site (in Figure 6 the call site of π_3 in π_4) an assignment must be moved into the callee, which, however, is prevented by the existence of another call site (in Figure 6 the call site of π_3 in π_2) because the semantics or performance would be affected (in Figure 6 the performance of paths calling π_3 via π_2 following the left branch in π_3). The uniformity constraint on start and end nodes is sufficient for detecting these situations and excluding the resulting anomalies. In the example of Figure 6, it prevents moving the assignment $dist(A) := cyclic(k)$ across the subprogram call of π_3 in π_4.

The second part of constraint $\mathcal{C}$ is important in order to avoid that it is unnecessarily restrictive. If the condition of the second part is satisfied, there is no interference with other call sites of the callee under consideration. Hence, movability depends (as for assignment patterns involving local variables) only on the context of the caller itself. This gives rise to consider it an *independence* condition.

Technically, it is important that constraint $\mathcal{C}$ can be integrated into the generic algorithm of the IDFA-framework of [13] computing the hoistability and sinkability information. Thus, we do not have to bother about it on the specification level of the analyses. It is automatically taken care of during the analysis using the modified generic algorithm.

4.2 A hierarchy of IDAP-algorithms

Having introduced constraint $\mathcal{C}$, we can now present the *high-level* specifications of our algorithms for interprocedural DAP, which match the pattern of their intraprocedural counterparts. As recalled above, DAP consists conceptually of PDCE and PRAE, which themselves rely on repeated applications of AS/DCE and AH/RAE, respectively; a repetition, which is necessary in order to capture all second-order effects between different assignment patterns. This basic algorithmic pattern can conveniently be expressed by means of regular-expression like terms (cf. [10]):

$$\mathsf{PDCE} \equiv (\mathsf{AS} + \mathsf{DCE})^+, \quad \mathsf{PRAE} \equiv (\mathsf{AH} + \mathsf{RAE})^+.$$

Using this notation our algorithms of *Pure* IDAP and *Full* IDAP, which differ in the assignments patterns considered, constitute the kernel of our hierarchy of IDAP-algorithms. They are defined as follows.

Basic IDAP-algorithms: Pure and Full IDAP.

- **Pure IDAP.** Like its intraprocedural counterpart, *Pure* IDAP focuses on distribution assignments which are of primary interest. This is indicated below by the subscript $\mathcal{D}$. The algorithm of Pure IDAP is then given by repeated applications of

$\mathsf{IPDCE}^{\mathcal{C}}_{\mathcal{D}}$ and $\mathsf{IPRAE}^{\mathcal{C}}_{\mathcal{D}}$ to distribution assignment patterns until the program stabilizes:

$$\mathsf{IDAP}_{pure} \equiv (\mathsf{IPDCE}^{\mathcal{C}}_{\mathcal{D}}\ \ \mathsf{IPRAE}^{\mathcal{C}}_{\mathcal{D}})^{+}.$$

The superscript $\mathcal{C}$ reminds here to the motion constraint imposed on the hoisting and sinking of global assignment patterns across subprogram calls (cf. Section 4.1). Hence, IPDCE and IPRAE expand to:

$$\mathsf{IPDCE}^{\mathcal{C}} \equiv (\mathsf{IAS}^{\mathcal{C}} + \mathsf{IDCE})^{+} \quad \text{and}$$

$$\mathsf{IPRAE}^{\mathcal{C}} \equiv (\mathsf{IAH}^{\mathcal{C}} + \mathsf{IRAE})^{+}.$$

- **Full** IDAP. In contrast to Pure IDAP, *Full* IDAP takes all assignment patterns into account. As pointed out in [10], this is necessary for also handling second-order effects introduced by interdependences between distribution assignments and ordinary assignments. For illustration consider a distribution specification $cyclic(exp)$, where exp denotes an arbitrary expression using scalar variables. The motion of distribution assignments matching this pattern is then blocked by every assignment modifying an operand of exp which can prohibit profitable optimizations. The algorithm of Full IDAP is thus given by:

$$\mathsf{IDAP}_{full} \equiv (\mathsf{IPDCE}^{\mathcal{C}}\ \ \mathsf{IPRAE}^{\mathcal{C}})^{+}.$$

In general, IDAP_{full} is more powerful than IDAP_{pure}, as it resolves second-order effects between all statements patterns, not only between distribution patterns. Its greater transformational power comes at the price of a greater computational complexity. This is the starting-point of further refinements of the basic algorithms, which as in the intraprocedural case results in a hierarchy of IDAP-algorithms of different power and efficiency allowing user-customized solutions.

Refinements of the basic IDAP-algorithms.

- *Enhancing power.* As in the intraprocedural case the transformational power of every of our IDAP-algorithms can be enhanced by replacing for ordinary assignments the procedure for partial dead-code elimination by the more powerful procedure for *partial faint-code elimination* (PFCE) (see [11] for details).

- *Enhancing efficiency.*

  (*1*) *Decoupling patterns.* The computational complexity of $\mathsf{IPDCE}^{\mathcal{C}}$ and $\mathsf{IPRAE}^{\mathcal{C}}$, and hence of IDAP depends above all on the number of iterations required for fully capturing the second-order effects; avoiding second-order effects by decoupling assignment patterns is thus a major means for enhancing the performance. This can be accomplished by means of a simple *preprocess*, which decouples distribution assignment patterns where possible. Intuitively, it replaces *alignments* by explicit distribution specifications and variables used in distribution specifications by constants, whenever possible (cf. [10]).

  In case the preprocess succeeds in eliminating even all sources of second-order effects, the following one-step sequence can equivalently be used instead of the corresponding basic algorithm above, where "X" stands for "pure" and "full", and "Y" for "$\mathcal{D}$" or "nothing", respectively:

$$\mathsf{IDAP}^{1\text{-}step}_{X} \equiv (\mathsf{IAS}^{\mathcal{C}}_{Y}\ \ \mathsf{IDCE}^{\mathcal{C}}_{Y})\ \ (\mathsf{IAH}^{\mathcal{C}}_{Y}\ \ \mathsf{IRAE}^{\mathcal{C}}_{Y})$$

  (*2*) *Limiting iterations.* The ratio underlying Pure IDAP is that distribution assignments are used in quite a restricted manner in practice only. Thus, it is likely to get a reasonable amount of the effect of Full IDAP (much) more efficiently. This suggests to limit the number of iterations performed, too. The extreme strategy resulting from this reasoning is a one-step heuristics considering only distribution patterns:

$$\mathsf{IDAP}^{1\text{-}step}_{pure} \equiv (\mathsf{IAS}^{\mathcal{C}}_{\mathcal{D}}\ \ \mathsf{IDCE}_{\mathcal{D}})\ \ (\mathsf{IAH}^{\mathcal{C}}_{\mathcal{D}}\ \ \mathsf{IRAE}_{\mathcal{D}})$$

We remark that the one-step heuristics applying the elementary transformations only once does not capture any second-order effect. However, it is extremely efficient and still reasonably effective in practice. Note that in the example of Figure 2 even the one-step heuristics yields the optimal result.

4.3 Implementation: The interprocedural DFAs and the transformations

In this section we demonstrate how to specify the interprocedural DFAs and the program transformations based thereof. They can directly be fed into the generic algorithm of the IDFA-framework of [13] returning the implementations of the IDFA-algorithms.

Interprocedural DFAs. Intraprocedurally, the DFA-problems underlying transformations like AH, RAE, AS, and DCE are usually specified in terms of an equation system, whose greatest solution yields the solution of the DFA-problem under consideration. In order to give a flavor of this specification style, we recall the equation systems for DCE and AH requiring both a backward analysis of the program under consideration. Here, n and m denote program points, and **LocHoist**, **LocBlock**, **Used** and **Mod** local predicates of edges indicating whether the assignment pattern under consideration occurs at the edge (n, m), or is blocked by the statement of this edge meaning that its left-hand side variable is used or modified, or one of its right-hand side operands is modified, or if the variable under consideration is used or modified by the statement of (n, m):[3]

$$dead_n \;=\; \bigwedge_{m \in succ(n)} \left(\neg \mathbf{Used}_{(n,m)} \;\wedge\; (dead_m \;\vee\; \mathbf{Mod}_{(n,m)}) \right)$$

$$hoistable_n \;=\; \bigwedge_{m \in succ(n)} \left(\mathbf{LocHoist}_{(n,m)} \;\vee\; (hoistable_m \;\wedge\; \neg \mathbf{LocBlock}_{(n,m)}) \right)$$

Intuitively, the greatest solutions of these equation systems denote the set of all variables and assignment patterns being dead at n and being hoistable to n, respectively. It is worth noting that the equation systems are "concrete" versions of the generic equation system characterizing the *maximal-fixed-point* (*MFP*) approach in the sense of Kam and Ullman [8]. They result from instantiating the generic equation system by a triple consisting of (1) a *lattice* $\mathcal{L}$, whose elements represent the data-flow information of interest, (2) a *local semantic functional* $[\![\ ]\!] : E \to (\mathcal{L} \to \mathcal{L})$, which specifies the effect of the elementary statements on the elements of $\mathcal{L}$, and a *start information* $l_0 \in \mathcal{L}$, which represents the information assumed to be valid on calling the procedure under consideration. E.g., the specification of the hoistability analysis is given by (1) the lattice $\mathcal{B}$ of Boolean truth values, (2) the local semantic functional $[\![\ ]\!]_{hst} : E \to (\mathcal{B} \to \mathcal{B})$ defined by $[\![\,(n, m)\,]\!](b) =_{df} (\, \mathbf{LocHoist}_{(n,m)} \;\vee\; (b \;\wedge\; \neg \mathbf{LocBlock}_{(n,m)}))$ for all $(n, m) \in E$, and (3) a start information out of $\mathcal{B}$ reflecting the assumption on the calling context of the procedure under consideration. Intraprocedurally, these triples can directly be fed into a generic algorithm computing the greatest fixed-point solution of the *MFP*-approach in the sense of Kam and Ullman accordingly. On the application level of the underlying intraprocedural DFA-framework, one does not have to know any details about this generic algorithm.

[3] Recall that we are considering edge-labeled flow graphs.

In our approach, which is based on the interprocedural DFA-framework of [13], this carries over to the interprocedural setting. Using this framework, the specifications of IAH, IRAE, IAS, and IDCE require in comparison to the intraprocedural case only a single component in addition, the so-called *return functional*. Intuitively, it is required for properly dealing with local variables of recursive procedures. In order to give a flavor of the interprocedural specifications and to illustrate the similarity to their intraprocedural counterparts, we present the specifications of IDCE and IAH. Like their intraprocedural counterparts, they can directly be fed into the generic algorithm of the IDFA-framework of [13], which (for the hoisting (sinking) analysis is modified in order to respect constraint $\mathcal{C}$), in order to compute the interprocedural counterpart of the *MFP*-solution.

Interprocedural deadness. The specification of the IDCE-analysis for some fixed variable x is as follows (cf. [13]):

1. *Lattice*: $(\, \mathcal{B},\, \wedge\,, \leq, \mathit{false}, \mathit{true}\,)$
2. *Local Semantic Functional* $[\![\ ]\!]_{dead} : E \to (\mathcal{B} \to \mathcal{B})$ defined by $\forall\, e \in E\; \forall\, b \in \mathcal{B}$:

   $$[\![\, e \,]\!]_{dead}(b) =_{df} \neg \mathbf{Used}_e \;\wedge\; (b \;\vee\; \mathbf{Mod}_e)$$

3. *Start Information*: $\mathit{false} \in \mathcal{B}$
4. *Return Functional*: $\mathcal{R}_{dead} \;:\; E_{call} \to (\, \mathcal{B}^2 \to \mathcal{B}\,)$ defined by $\forall\, e \in E_{call}\; \forall\, (b_1, b_2) \in \mathcal{B}^2$:

   $$\mathcal{R}_{dead}(e)(b_1, b_2) =_{df} \begin{cases} b_2 & \text{if } \mathit{PotAcc}\,(\mathit{callee}(\mathrm{e})) \\ \mathbf{Used}_e \;\vee\; b_1 & \text{otherwise} \end{cases}$$

Note that the first three components coincide with their intraprocedural counterparts. New is the fourth component, the *return functional*. Intuitively, it extracts from the data-flow information being valid immediately before entering the called subprogram (available as b_1), and the data-flow information being valid immediately before returning from it (available as b_2), the data-flow information which is valid immediately afterwards. Technically, this is accomplished in that the generic computation procedure of the underlying IDFA-framework works (implicitly) on a DFA-stack of data-flow informations, which mimics the run-time stack of a run-time system. However, on the application level of the framework, one does not have to bother with these details, which can be found in [13]. Here, we only remark that the predicate

$PotAcc$ indicates wether the location of the variable x under consideration can be accessed by the callee, an information, which in case that x is passed as a reference parameter is computed by a preprocess. Otherwise, it reduces to whether x is a global or local variable.

Interprocedural hoistability. Next, we present the specification of the IAH-analysis for some fixed assignment pattern α. Intuitively, α is *interprocedurally hoistable* to a program point n, if on every program path leaving node n the first use or modification of its left-hand side variable or the first modification of one of its right-hand side operands is preceded by an occurrence of α. Formally, this is captured by:

1. *Lattice*: $(\mathcal{B}^2, \wedge, \leq, (\mathit{false}, \mathit{false}), (\mathit{true}, \mathit{true}))$

2. *Local Semantic Functional*: $[\![\ ]\!]_{hst} : E \to (\mathcal{B}^2 \to \mathcal{B}^2)$ defined by

   $$\forall e \in E \ \forall (b_1, b_2) \in \mathcal{B}^2 .\ [\![e]\!]_{hst}(b_1, b_2) =_{df} (b_1', b_2')$$

   where

   $$b_1' =_{df} A\text{-}LocHoist_e \vee (b_1 \wedge \neg A\text{-}LocBlock_e)$$

   $$b_2' =_{df} \begin{cases} b_2 \wedge NoGlbActBlck_e & \text{if } e \in E \backslash E_r \\ \mathit{true} & \text{otherwise} \end{cases}$$

3. *Start Information*: $(\mathit{false}, \mathit{true}) \in \mathcal{B}^2$

4. *Return Functional*: $\mathcal{R}_{hst} : E_{call} \to (\mathcal{B}^2 \times \mathcal{B}^2 \to \mathcal{B}^2)$ defined by

   $$\forall e \in E_{call} \ \forall ((b_1, b_2), (b_3, b_4)) \in \mathcal{B}^2 \times \mathcal{B}^2 :$$

   $$\mathcal{R}_{hst}(e)((b_1, b_2), (b_3, b_4)) =_{df} (b_5, b_6)$$

   where

   $$b_5 =_{df} \begin{cases} b_3 & \text{if } PotAcc(callee(e)) \\ b_1 \wedge b_4 & \text{otherwise} \end{cases}$$

   $$b_6 =_{df} b_2 \wedge b_4$$

In contrast to the IDCE-specification, the lattice required here is a product lattice. The first component is the counterpart of the lattice intraprocedurally used; the second component is required for the proper treatment of assignment patterns involving global and local variables. Note that the effect of a subprogram call to a global variable must be maintained after returning from a call, whereas the effect to a local variable must be reset to the state before calling the subprogram. The second component of the product lattice gives us the handle to keep track on modifications of items involving local and global variables. For the IDCE-analysis this is not necessary, as variables are always "local" or "global", i.e., "mixed" situations as for assignment patterns composed of global and local variables are impossible.

Similar to the IDCE-specification, also the IAH-specification indicates a simple way of how to handle reference parameters in our approach. This can be accomplished by means of a preprocess computing alias-information for variables. The information computed can then be exploited in a black-box fashion simply by modifying the local predicates involved; a fact, which in the specification above is reflected by prefixing the names of local predicates by "A-", indicating that information on "may-aliases" and "must-aliases" of the items under consideration is taken into account. This has been described in detail in [13].

The transformations. The transformations induced by the solutions of the IDFA-problems for AH, RAE, AS, and DCE are essentially as in the intraprocedural case. For the elimination transformations (DCE and RAE), this means to eliminate all occurrences which have been identified as dead or redundant, respectively. For the motion analyses (AH and AS), this means to replace all original occurrences of the assignment pattern under consideration by new occurrences inserted at program points where the motion process is blocked by some assignment or which have a predecessor (in case of AH) or a successor (in case of AS) which does not satisfy the motion predicate under consideration. First practical experiences with our IDAP-approach are very promising as outlined in Section 1, and emphasize its effectivity and power.

5 Conclusions

DAP and IDAP rely conceptually on the very same intuitions. However, we demonstrated that a naive interprocedural extension of the intraprocedural approach fails by suffering from severe anomalies. We showed how to overcome these anomalies by imposing a natural and sufficient constraint on moving occurrences of global assignment patterns across subprogram calls. As in the intraprocedural case this led to a *practical* and *powerful* approach for IDAP, which enjoys the central features of its intraprocedural counterpart, in particular, its *flexibility* allowing a hierarchy of IDAP-algorithms of varying power and efficiency offering user-customized solutions. Moreover, all modifications required for overcoming the motion anomalies could be encapsulated inside the generic algorithm of the interprocedural DFA-framework underlying our approach. Thus, one does not have to bother with them at the application level. First practical experiences underline the power and importance of our IDAP-approach. An implementation of the complete approach within the VFCS system [2] is in progress.

References

[1] G. Agrawal, J. Saltz, and R. Das. Interprocedural partial redundancy elimination and its application to distributed memory compilation. In *Proc. ACM SIGPLAN '95 Conf. Prog. Lang. Design and Impl.* (*PLDI'95*), pages 258–269, La Jolla, CA, 1995.

[2] 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*. Institute for Software Technology and Parallel Systems, University of Vienna, Vienna, 1996.

[3] F. Coelho and C. Ancourt. Optimal compilation of HPF remappings. Technical Report TR CRI A-277, Ecole des Mines de Paris, Centre de Recherche en Informatique, October 1995.

[4] High Performance Fortran Forum. High Performance Fortran language specification version 2.0. Technical report, Rice University, Houston,TX, January 1997.

[5] 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.

[6] 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.

[7] 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, MN, 1992.

[8] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:309 – 317, 1977.

[9] K. Kennedy and A. Sethi. A constraint based communication placement framework. Technical Report CRPC-TR95515-S, Department of Computer Science, Rice University, Houston,TX, February 1995.

[10] J. Knoop and E. Mehofer. Optimal distribution assignment placement. In *In Proc. of Euro-Par '97*, LNCS 1300, pages 364 – 373, Passau, Germany, 1997. Springer-Verlag.

[11] J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. In *Proc. ACM SIGPLAN '94 Conf. Prog. Lang. Design and Impl.* (*PLDI'94*), pages 147–158, Orlando, FL, 1994.

[12] J. Knoop, O. Rüthing, and B. Steffen. The power of assignment motion. In *Proc. ACM SIGPLAN '95 Conf. Prog. Lang. Design and Impl.* (*PLDI'95*), pages 233–245, La Jolla, CA, 1995.

[13] J. Knoop, O. Rüthing, and B. Steffen. Towards a tool kit for the automatic generation of interprocedural data flow analyses. *Journal of Programming Languages*, 4(4):211–246, 1996.

[14] E. Mehofer and H. Zima. Distribution assignment placement. Technical Report TR 96-5, Institute for Software Technology and Parallel Systems, University of Vienna, Austria, 1996.

[15] D.J. Palermo, E.W. Hodges, and P. Banerjee. Interprocedural array redistribution data-flow analysis. In *Proc. 9th Workshop on Lang. and Comp. for Parallel Computing*, San Jose, CA, 1996.

[16] S. Ramaswamy and P. Banerjee. Automatic generation of efficient array redistribution routines for distributed memory multicomputers. In *Proc. 5th Symp. Frontiers of Massively Parallel Computation*, pages 342–349, McLean, VA, 1995.

[17] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Conf. Rec. 15th ACM Symp. Principles of Prog. Lang.* (*POPL'88*), pages 2 – 27, San Diego, CA, 1988.

[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.