Distribution Assignment Placement: Effective Optimization of Redistribution Costs *

Jens Knoop †   Eduard Mehofer ‡

Abstract

Data locality and workload balance are key factors for getting high performance out of data-parallel programs on multiprocessor architectures. Data-parallel languages like High Performance Fortran (HPF) thus offer means for specifying data distributions as well as allow distributions to be dynamically changed in order to maintain these properties. Redistributions, however, can be quite expensive and significantly degrade a program's performance. Here, we present a novel, aggressive approach for avoiding unnecessary remappings which works by eliminating *partially dead* and *partially redundant* distribution changes. Basically, this approach evolves from extending and combining two algorithms for these optimizations achieving each on its own optimal results. Combining them requires a refined optimality investigation. Moreover, the data-parallel setting leads to a hierarchy of algorithms of varying power and efficiency allowing user-customized solutions. The power and flexibility of the new approach are demonstrated by illustrating examples. Practical experiences underline its importance and effectivity.

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

1 Motivation

The user-controled distribution of data across the local memories of processing nodes is a central feature of data-parallel languages like *High Performance Fortran* (HPF) [7], *Fortran D* [8],

---

*A preliminary version of this article was presented at Euro-Par'97 (cf. [15]).

†Universität Passau, Fakultät für Mathematik und Informatik, Innstraße 33, D-94032 Passau, Germany (e-mail: knoop@fmi.uni-passau.de).

‡Universität Wien, Institut für Softwaretechnik und Parallele Systeme, Liechtensteinstraße 22, A-1090 Vienna, Austria (e-mail: mehofer@par.univie.ac.at).

or *Vienna Fortran* [27]. Since many scientific applications typically consist of different computational phases owning each a best data distribution, dynamic redistributions have proven useful in maintaining data locality and workload balance. Further redistributions have successfully been employed for kernels such as parallel linear algebra routines [24], 2D-FFT (two-dimensional Fast Fourier Transform) [11], or ADI (Alternating Direction Implicit) [21], a partial differential equation (PDE) solver. In HPF redistributions are supported by the procedure interface and by executable directives (REALIGN/REDISTRIBUTE). But remappings can be quite expensive as communication is required to migrate the array elements to their new owning processors. This is important because even well-written HPF programs can result in unnecessary redistributions as demonstrated in Figure 1. This can significantly degrade a program's performance. Avoiding unnecessary redistributions is of key importance for gaining efficiency and improving performance.

In this article we present a novel, aggressive approach for *distribution assignment placement* (DAP), which, in essence, works by eliminating *partially dead* and *partially redundant* distribution changes. Basically, this approach evolves from extending and combining two algorithms for partially dead and partially redundant assignment elimination for sequential programs achieving optimal results (cf. [18, 19]). Intuitively, the new algorithm computes beneficial insertion points for distribution assignments by means of code *hoisting* and *sinking* interleaved by eliminating *redundant* and *dead* code, which uniformly captures removal of unnecessary remappings in straight-line code as well as in loops. Besides the well-known *second-order effects* (cf. Section 2) introduced by interdependences of different statement patterns, which as usual can be overcome by repeated applications of the elementary transformations, the interleaving of all four elementary transformations reveals additional interdependences of the transformations themselves: the final result depends on the particular sequence of the elementary transformations requiring a refined optimality consideration. On the other hand, and in contrast to the sequential setting, the data-parallel setting leads to a hierarchy of algorithms of varying power and performance allowing user-customized solutions. While the basic version of *Pure* DAP focuses on distribution assignments and resolves second-order effects between them, the algorithm of *Full* DAP resolves second-order effects between all statements, i.e., ordinary and distribution assignments. Partially dead and partially redundant assignments remaining in the program cannot be eliminated further by means of the elementary transformations without changing its branching structure or impairing some of its executions.

Structure of the article. After demonstrating the power and flexibility of our approach by illustrating examples in Section 2, we discuss related work in Section 3, and subsequently summarize the basic notions required for the formal development of our approach in Section 4. Central is then Section 5, where we stepwise develop our algorithms of Pure DAP and Full DAP, and subsequently construct a variety of practically important variants of these algorithms resulting in a hierarchy of DAP-algorithms of varying power and performance. In Section 6 we then report on practical experiences underlining the importance of DAP, and draw our conclusions in Section 7. The Appendix, finally, contains complementary material, in particular an extended example demonstrating the impact of second-order effects between eliminating partially dead and partially redundant assignments.

2 Illustrating Examples

HPF allows programmers to change the distribution of arrays explicitly with *redistribute/realign directives* or implicitly at subprogram boundaries. Whereas the redistribution of an array causes all aligned arrays to be remapped as well, realigning an array does not affect any other data object. *Distribution assignments* as proposed in [22] allow us a uniform treatment of all distribution changes occurring in a program. They are generated by the compiler whenever an array is associated with a distribution, and are part of our intermediate program representation.

A small example. As recalled above, reducing the number of remappings at run-time is of key importance in order to gain efficiency. The elimination of (totally) *redundant* and *dead* distribution assignments is in general insufficient as illustrated by the example of Figure 1 showing a well-written HPF program. The arrays A and B are assumed to be statically distributed by BLOCK, and subroutine F is assumed to not modify these distributions. The distribution assignments to A and B with BLOCK and CYCLIC as right-hand side terms indicate that their distribution is changed to BLOCK and CYCLIC, respectively, at the point of their occurrences. They are inserted by the compiler to establish the requested distributions on entry and to restore the original ones on exit according to the HPF semantics [22]. Note that none of the distribution assignments is (totally) redundant or dead. This prevents to simply eliminate them. However, the distribution assignments after leaving subroutine F are *partially dead* because they are dead within the loop (and alive with respect to node **3**). Thus, they can be removed from the loop as shown in Figure 1(b). As a side-effect, this turns the totally live distribution assignments before entering subroutine F, which were originally required on every program execution, to be *partially redundant*. They are redundant within the loop (but not redundant with respect to the static distribution) and can thus

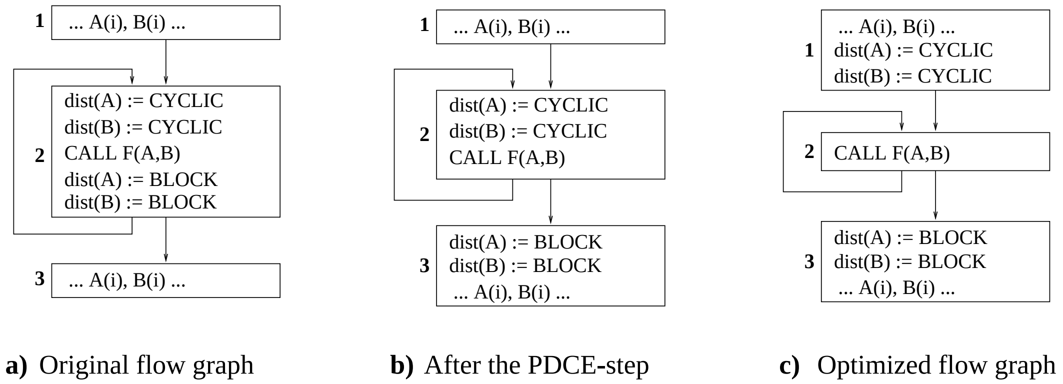
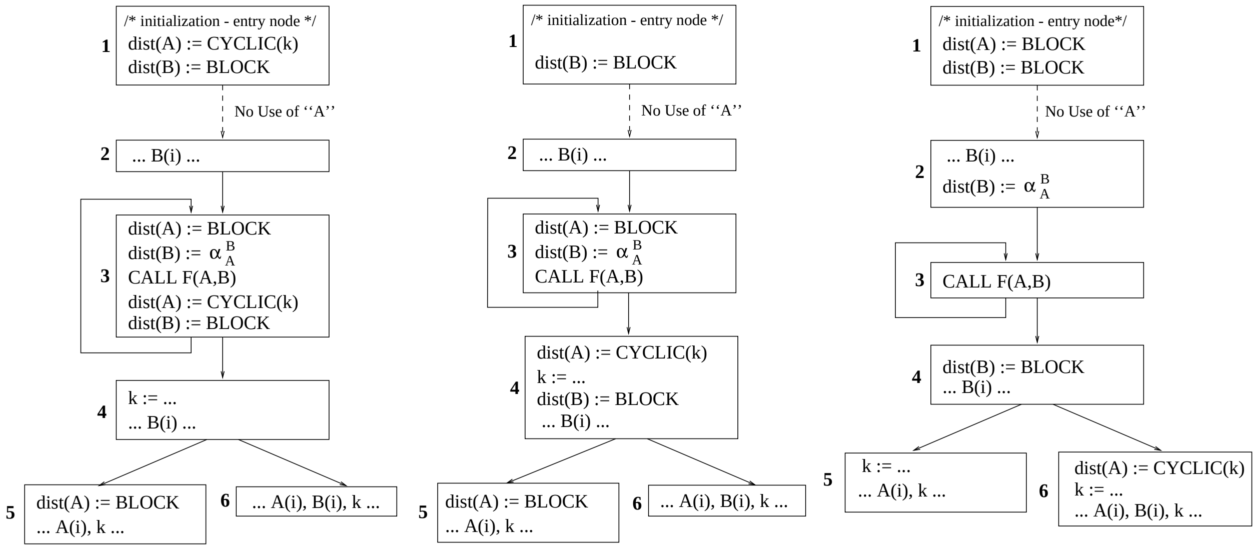


Figure 1: Small example.

be removed from the loop too as shown in Figure 1(c). Such effects are usually called *second-order effects* (cf. [26]).

A complex example. The example of Figure 2 illustrates the full power of our approach to resolve second-order effects between different statement patterns. The use of alignments for specifying the distribution of arrays introduces dependences between distribution assignments. Parameters in distribution specifications like in BLOCK(i) or CYCLIC(i) introduce dependencies between ordinary and distribution assignments, as well. Both kinds of dependencies are in general the source of second-order effects as illustrated in Figures 2(a), 2(b), and 2(c). As before, a subroutine which is transparent for the distributions of arrays A and B is called within a loop. Array B is aligned to array A on entry of subroutine F, denoted by $\alpha_{\mathsf{A}}^{\mathsf{B}}$. Analogously to Figure 1, the distribution assignments at the exit of subroutine F can be removed from the loop by placing them in node **4** as they are dead inside the loop (cf. Figure 2(b)). Whereas the distribution assignment dist(B) := BLOCK at node **4** is now totally live, dist(A) := CYCLIC(k) is still partially dead. Moving it to node **6**, however, where it would be totally live as well, is prevented by the assignment to variable k (cf. Figure 2(b)). Hence, moving the ordinary assignment to k first suspends this blockade and subsequently allows us to eliminate the partially dead assignment dist(A) := CYCLIC(k) as desired. As a side-effect this turns the distribution assignment at node **5** to be totally redundant, which can thus be eliminated as well. As another second-order effect, the distribution assignment to A at the entry of the loop becomes partially redundant, and can be hoisted out of the loop to node **1**. This suspends the blockade of the alignment dist(B) := $\alpha_{\mathsf{A}}^{\mathsf{B}}$ at the entry of the loop too, which, as another second-order effect, can now be moved to node **2**.

Figure 2(c) shows the result of the complete transformation, which is achieved by our approach. Note that the optimized flow graph is optimal: it is free of any partially dead and partially redundant distribution assignment.



a) Original flow graph **b)** After the PDCE-step **c)** Optimized flow graph

Figure 2: Complex example: Illustrating the impact of second-order effects.

3 Related Work

Redistribution analysis has been addressed by several researchers. Hall et al. [12] presented techniques for hoisting remappings out of loops and eliminating dead remappings. Their approach is incomparable to ours as it takes interprocedural information into account, but does not consider the more general problem of eliminating partially dead and partially redundant remappings. 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. Optimality in their sense means that for a given remapping, a minimal number of messages is sent over the network. The problem of reducing the overall number of remappings by employing code motion has not been addressed. Similarly, this holds for the approach of Ramaswamy et al. [25], whose focus lies on the automatic generation of efficient routines for migrating the array elements to their new owning processors. Palermo et al. [23] present several analyses related to dynamic redistributions: computation of reaching distributions, making all remappings (including the implicit ones at subprogram boundaries) explicit and removing the redundant ones, and converting programs with dynamically distributed arrays into subset HPF. Motion of data remappings, however, is not considered.

Communication optimization frameworks relying on code motion techniques for *partial redundancy elimination* (PRE) have been proposed by Gupta et al. [10], Agrawal et al. [1], and Kennedy et al. [13]. Gupta et al. applied PRE-techniques to available section descriptors in order to reduce communication costs, whereas Agrawal et al. focused on the problem of interprocedural placement of communication statements. Both approaches are based on the bidirectional PRE-framework of Dhamdhere et al. [6]. In contrast, the constraint-based communication placement framework proposed by Kennedy et al. is based on the unidirectional PRE-algorithm for *lazy code motion* of [16].

As demonstrated above, PRE-techniques are insufficient for DAP, which requires both the elimination of partially dead and partially redundant instructions. These problems have been separately investigated in [18] and [19] for a sequential setting. Here, we show how to enhance and combine these algorithms in order to arrive at a hierarchy of DAP-algorithms for a data-parallel setting.

4 Preliminaries

As usual we represent programs by *directed flow graphs* $G=(N,E,\mathbf{s},\mathbf{e})$ with node set N and edge set E. Nodes $n \in N$ represent *basic blocks* of instructions ι, edges $(m,n) \in E$ the nondeterministic branching structure of G, and $\mathbf{s}$ and $\mathbf{e}$ its unique *start node* and *end node*, which are assumed to be free of incoming and outgoing edges, respectively. Additionally, $succ(n) =_{df} \{ m \mid (n,m) \in E \}$ and $pred(n) =_{df} \{ m \mid (m,n) \in E \}$ denote the set of all immediate successors and predecessors of a node n. All nodes of G are assumed to lie on a path from $\mathbf{s}$ to $\mathbf{e}$. Moreover, all statements occurring in a program are classified as follows: (ordinary) *assignment statements* involving both scalar and indexed variables; the *empty statement* skip; *distribution assignments* of the form $\mathsf{dist(A)} := \delta$ with A denoting an array and δ an arbitrary distribution, which are generated by the compiler and uniformly express distribution changes occurring throughout the program at subprogram boundaries and for redistribute/realign directives; subprogram calls, and output operations of the form $out(t)$ forcing all operands of term t to be alive.[1] Finally, all edges starting in a node with more

[1] In practice, conditions in if-statements and assignments to global variables (variables whose declaration is outside the scope of the flow graph under consideration) must be treated analogously. As demonstrated in Appendix B, this is straightforward in our approach.

than one successor and leading to a node with more than one predecessor are assumed to be split by a synthetic node. This is typical for code motion transformations in order to avoid that the motion process is blocked (cf. Appendix C).

5 Distribution Assignment Placement

In this section we stepwise develop our hierarchy of DAP-algorithms providing user-customized solutions for eliminating unnecessary overhead due to distribution changes. In essence, this works by exploiting the trade-off between efficiency and power of the transformation. The kernel of this hierarchy is constituted by the algorithms of *Pure* DAP and *Full* DAP, which we present first. For convenience, we consider an arbitrary but fixed flow graph G. Moreover, we denote the set of ordinary and distribution assignment patterns occurring in G by $\mathcal{P}$, and its subset of distribution patterns by $\mathcal{D} \subseteq \mathcal{P}$.

As illustrated in Section 2 the essence of DAP is to avoid unnecessary executions of distribution assignments at run-time. Intuitively, a distribution assignment is *unnecessary*, if it is *dead*, i.e., there is no program continuation on which its left-hand side variable is used without a preceding distribution assignment, or if it is *redundant*, i.e., on every program path reaching it a distribution assignment of the same pattern has been executed without an intermediate distribution change. Thus, DAP relies on the combined effects of

- eliminating *partially dead* and *partially redundant* assignments.

This is important because both subproblems can optimally be solved as it was demonstrated in [18] and [19] presenting algorithms for *partially dead code elimination* (PDCE) and *partially redundant assignment elimination* (PRAE), respectively. Here, we are going to show how to enhance these algorithms originally designed for a standard sequential program setting to the data-parallel one, and how to combine them uniformly in order to arrive at a hierarchy of user-customized DAP-algorithms.

5.1 Partially Dead Code Elimination

Like DAP, PDCE consists conceptually of two elementary transformations: *assignment sinkings* (AS) and *dead code eliminations* (DCE). Assignment sinkings move assignments as far as possible in the direction of the control flow (i.e., while maintaining the program semantics). Intuitively, this places them in a context as *specific* as possible, and maximizes the potential of dead code, which subsequently is eliminated by a dead variable analysis. Table 1 summarizes the data flow analyses for assignment sinkings and dead assignment eliminations, which uniformly capture ordinary and distribution assignments. In the following we discuss the intuition underlying both analyses.

Basically, the AS-analysis is controled by the local predicates **LocBlock** and **LocSink**, which are the handle for restricting assignment sinkings to *admissible* ones, i.e., semantics preserving ones. In essence, admissibility requires that assignments are never moved across instructions *blocking* them (expressed by **LocBlock**), i.e., using or modifying their left-hand side variables, or modifying some of their right-hand side variables complemented by variables used in index expressions of their left-hand side variables. Additionally, subprogram calls are considered blockades for assignments as we do not perform an interprocedural analysis. In essence, the same blocking constraints we impose on distribution assignments: distribution assignments of the form $\mathsf{dist(A)} := \delta$ are blocked by any occurrence of A and any definition of a variable used in δ. In particular, this guarantees that assignment sinkings respect the distribution proposed by the programmer: for each array reference the distributions in the original and the optimized program are identical. Note that this constraint could be weakened according to the quite typical assumption for data-parallel languages that array distributions do not affect the program semantics. In our approach, this can easily be achieved by defining the predicate **LocBlock** for ordinary and distribution assignments differently. The second predicate, **LocSink**, identifies all assignment occurrences that can be moved at all: *sinking candidates* are occurrences of ordinary as well as distribution assignments inside a basic block which are *downwards exposed*, i.e., not blocked by a subsequent assignment of the same basic block.

Elimination of dead assignments is based on dead variable analysis which is controled by the local predicates **Used** and **Mod**. An occurrence of an ordinary assignment with left-hand side variable x is dead, if x is dead, i.e., on every program continuation every right-hand side occurrence of x complemented by occurrences of x in left-hand side index expressions (expressed by **Used**) is preceded by a modification of x (expressed by **Mod**). For distribution assignments the local predicates have to be adapted accordingly. The distribution of an array A is accessed by every ordinary assignment with left-hand side A and every right-hand side occurrence of A complemented by occurrences of A in left-hand side index expressions. Modifications of distributions are accomplished

Assignment Sinking Analysis (AS): (Let $\alpha \in \mathcal{P}$)

- $\mathbf{LocSink}_n(\alpha)$: There is a sinking candidate of α in n.
- $\mathbf{LocBlock}_n(\alpha)$: The sinking of α is blocked by some instruction of n.

$$\begin{aligned}
\textbf{N-SINK}_n &=_{df} \begin{cases} \mathit{false} & \mathit{if} \quad n = \mathbf{s} \\ \prod\limits_{m \in pred(n)} \textbf{X-SINK}_m & \mathit{otherwise} \end{cases} \\
\textbf{X-SINK}_n &=_{df} \mathbf{LocSink}_n + \textbf{N-SINK}_n * \overline{\mathbf{LocBlock}_n}
\end{aligned}$$

Insertion Points:[a] [b]

$$\begin{aligned}
\textbf{N-INSERT}_n &=_{df} \textbf{N-SINK}^{\star}_n * \mathbf{LocBlock}_n \\
\textbf{X-INSERT}_n &=_{df} \textbf{X-SINK}^{\star}_n * \begin{cases} \mathit{true} & \mathit{if}\, n = \mathbf{e} \\ \sum\limits_{m \in succ(n)} \overline{\textbf{N-SINK}^{\star}_m} & \mathit{otherwise} \end{cases}
\end{aligned}$$

Dead Variable Analysis ($\equiv$ Dead Code Elimination DCE):[c]

- $\mathbf{Used}_\iota(x)$: There is a use of the value/distribution of x in ι.
- $\mathbf{Mod}_\iota(x)$: The left-hand side of the instruction ι changes the value/distribution of x.

$$\begin{aligned}
\textbf{N-DEAD}_\iota &=_{df} \neg\mathbf{Used}_\iota \; * \; (\textbf{X-DEAD}_\iota \; + \; \mathbf{Mod}_\iota) \\
\textbf{X-DEAD}_\iota &=_{df} \prod_{\hat{\iota} \in succ(\iota)} \textbf{N-DEAD}_{\hat{\iota}}
\end{aligned}$$

Elimination Points:[a] $\quad \mathbf{ELIM}_\iota =_{df} \textbf{X-DEAD}^{\star}_\iota$

Table 1: Assignment Sinking and Dead Variable Analysis

by distribution assignments.

The PDCE-Algorithm and its Optimality. Following [18] the second-order effects induced by interdependences of different assignment patterns (cf. Section 2) are fully captured by repeatedly applying the elementary transformations of PDCE until the program stabilizes. This is conveniently

[a] $\textbf{N-SINK}^{\star}$, $\textbf{X-SINK}^{\star}$, and $\textbf{X-DEAD}^{\star}$ denote the greatest solutions of the equation systems for sinkability and deadness, respectively.

[b] This means to replace all sinking candidates of the assignment pattern under consideration by instances of this pattern inserted at program points satisfying **N-INSERT** or **X-INSERT**.

[c] For the ease of presentation the equations are specified at instruction level.

expressed by means of the following regular-expression like term

$$\mathsf{PDCE} \equiv (\mathsf{AS} + \mathsf{DCE})^+$$

where AS and DCE denote a single application of the assignment sinking and dead assignment elimination procedure to all assignment patterns of $\mathcal{P}$. As proved in [9] the program finally resulting from PDCE is independent of the specific order of the elementary transformations: it is uniquely determined up to (irrelevant) local reorderings in basic blocks. Moreover, it is *best* (*optimal*): it is *better* than any other program in the set of programs $\mathcal{G}_{pcde} =_{df} \{G' \mid G \vdash^*_{(\mathsf{AS+DCE})} G'\}$ derivable from G by means of sequences of admissible assignment sinkings and dead code eliminations. Central is here the precise meaning of "better": a program $G' \in \mathcal{G}_{pcde}$ is *better* than a program $G'' \in \mathcal{G}_{pcde}$ if and only if for every assignment pattern the number of assignments executed on every path in G' is less or equal to that in G'' (cf. [9, 18]). This is important because it directly implies that the program resulting from PDCE is best whatever the actual execution costs of the occurrences of a specific assignment pattern may be. As we are going to show below, for DAP it is necessary to take execution costs of assignment patterns into account.

5.2 Partially Redundant Assignment Elimination

PRAE is conceptually dual to PDCE: *assignment hoistings* (AH) move assignments as far as possible in the opposite direction of the control flow in order to place them in a context as *universal* as possible. Intuitively, this maximizes the potential of redundant code, which subsequently is eliminated by a *redundant assignment elimination* (RAE). Due to the duality to PDCE, we present the analyses for AH and RAE in Appendix A only (see Table 2). We just remark that the local predicate **LocBlock** for assignment hoisting is defined like its counterpart for assignment sinking. Similarly, this also holds for the local predicate **LocHoist** identifying *hoisting candidates*: these are occurrences of ordinary and distribution assignments inside a basic block which are *upwards exposed*, i.e., not blocked by a preceding instruction of the same basic block.

The PRAE-Algorithm and its Optimality. Like for PDCE, also for PRAE repeated applications of assignment hoistings and redundant assignment eliminations suffice to completely capture the second-order effects between different statement patterns, i.e.

$$\mathsf{PRAE} \equiv (\mathsf{AH} + \mathsf{RAE})^+.$$

The program finally resulting from PRAE is uniquely determined up to local reorderings in basic blocks. Moreover, it is *best* (*optimal*) in the set of programs $\mathcal{G}_{prae} =_{df} \{G' \mid G \vdash^*_{(\mathsf{AH+RAE})} G'\}$, which are derivable from G by sequences of admissible assignment hoistings and redundant assignment eliminations with respect to the relation "better" as defined in Section 5.1 (cf. [19]).

5.3 Interdependences between PDCE and PRAE

As recalled above, the elementary transformations of PDCE and PRAE can be applied in any order without affecting the program finally resulting (cf. [18, 19]). In fact, in the set of programs derivable by admissible assignment sinkings (hoistings) and dead (redundant) assignment eliminations, there is a "globally best" one, which finally is reached by every transformation sequence. Unfortunately, this property gets lost as soon as all four elementary transformations are interleaved as required for DAP. This is illustrated by the flow graphs of Figure 3. Applying PDCE first we arrive at the program of Figure 3(b), applying PRAE first we arrive at the program of Figure 3(c). Note that both programs are invariant under further admissible assignment motions and dead (redundant) assignment eliminations. Moreover, they are incomparable. While each path through the program fragment of Figure 3(c) contains precisely one distribution assignment, there is a path through the fragment of Figure 3(b) being free of distribution assignments, and another one containing two.

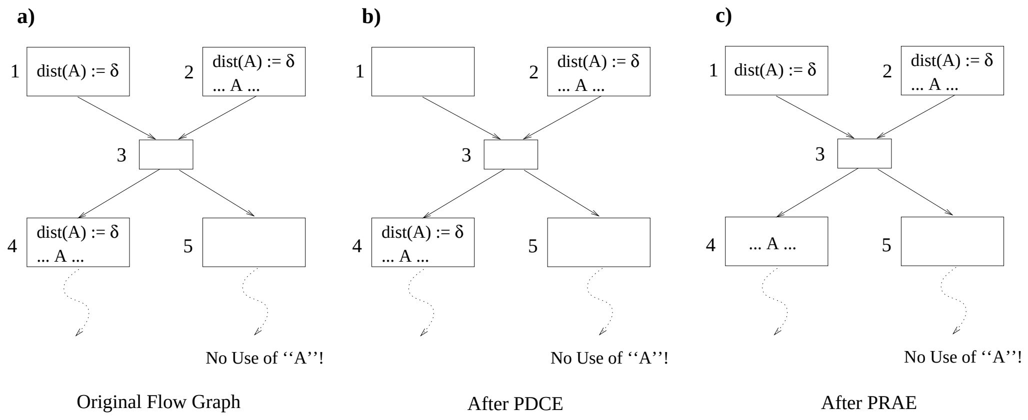


Figure 3: Interdependences between PDCE and PRAE.

As pointed out by this example, PDCE and PRAE influence each other by mutually removing opportunities for their respective counterpart. In fact, we conjecture that like in this example "maximally transformed" programs are either identical (up to irrelevant reorderings in basic blocks) or incomparable with respect to the order of Section 5.1 counting assignments for every assignment pattern separately. But programs may be considered incomparable for which an order according to an intuitive notion of "better" exists as demonstrated in the example of Figure 4. The programs of Figure 4(b) and 4(c) result from program 4(a) by beginning with PRAE and PDCE, respectively. On paths starting from node **1** program 4(c) contains less remappings than program 4(b). On path (3,4,5,7) the same remappings are required for both versions. On path (3,4,5,6), however, the distribution assignments dist(A) := BLOCK(i) and dist(B):= $\alpha_{\mathsf{A}}^{\mathsf{B}}$ are executed in 4(b), whereas in 4(c) the assignment dist(A) := BLOCK(i) is performed twice. Hence, both programs are incomparable wrt the relation "better" of Section 5.1. However, taking execution costs into account, both programs would perfectly be comparable, if the execution costs of dist(A) := BLOCK(i) and dist(B):= $\alpha_{\mathsf{A}}^{\mathsf{B}}$ coincide. Then, the program of 4(c) is strictly "better" than that of 4(b); a fact, which is not reflected by the order of Section 5.1. Unfortunately, validating the equality of costs of different distribution assignment patterns is in general not trivial and requires a performance prediction evaluating program paths statically. Thus, it is worth noting that in program 4(b) both remappings on the path from node **3** to node **6** have to be done in any case, while on the corresponding path of the program of 4(c) the second remapping is redundant. By means of *distribution assignment masking* (cf. Section 5.6) this unnecessary remapping can be suppressed at the price of a much cheaper run-time test implying that program 4(c) is "better" than program 4(b) without having to refer to the relative costs of dist(A) := BLOCK(i) and dist(B):= $\alpha_{\mathsf{A}}^{\mathsf{B}}$. Note, however, that "better" relies now on a specific code generation strategy.

In spite of these subtleties, we have that the process of interleaving all four elementary transformations of PDCE and PRAE always comes up with a program, which cannot be improved any further by means of semantics preserving eliminations of partially redundant and partially dead assignments leaving the program structure invariant, i.e., with a program being "locally best". In the following, we thus concentrate on the question in which order to apply the elementary transformations in order to achieve fast stabilization. Obviously, AS- and AH-steps should always be interleaved with DCE- and RAE-steps (as AS and AH essentially reverse each other's effect). This implies interleaving of PDCE and PRAE which, in general, must be applied repeatedly in order to reach a stable state (see Appendix D for an extensive example). In essence, this is a consequence of the fact that RAE (DCE) removes blockades which prevent partially dead (partially redundant) code to be sunk (hoisted) to places where it becomes totally dead (redundant). Thus, the question reduces essentially to that of starting with PDCE or PRAE. The following facts suggest that starting with PDCE is in general superior. First, the situation displayed in the example of Figure 1,

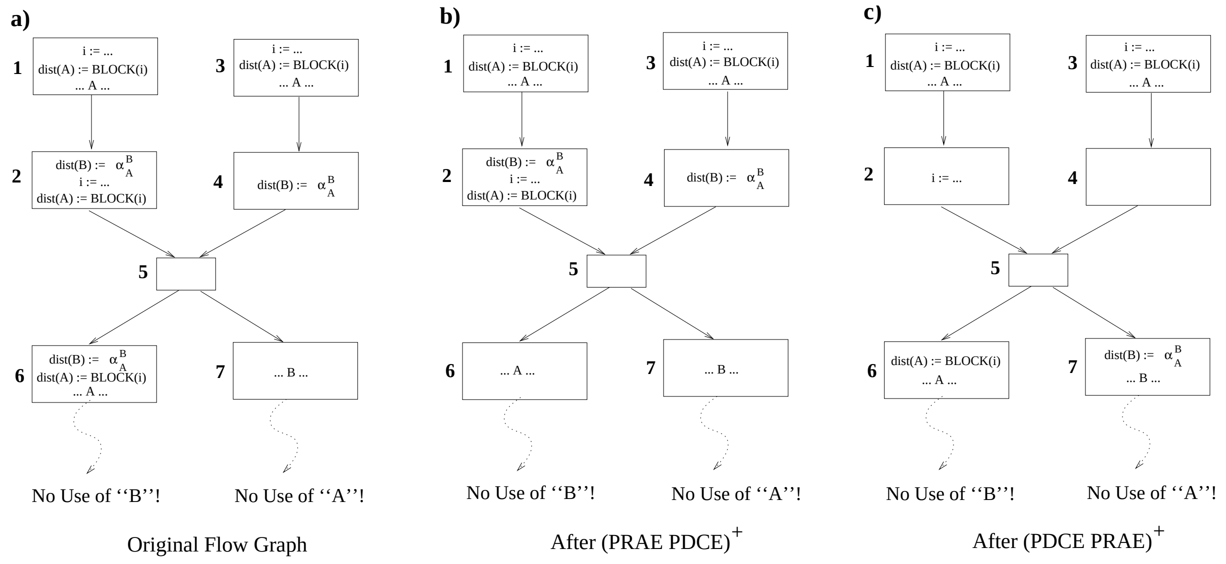


Figure 4: "Maximally" transformed programs are sometimes comparable.

where PDCE is a prerequisite for enabling PRAE, can be considered quite typical for data-parallel programs. Second, there is an important difference between PDCE and PRAE as stated in Lemma 1.

Lemma 1 (PDCE – PRAE **Interplay**)

1. PRAE *never creates (partially) dead assignments.*
2. PDCE *may create (partially) redundant assignments.*

Thus, in spite of their conceptual duality, the effects and hence the interplay of PDCE and PRAE are not completely dual. Though not sufficient, starting with PDCE enlarges the chance that less iterations are required as demonstrated in Section 2 where already PDCE followed by PRAE terminates with a locally best program.

5.4 Pure DAP

Pure DAP focuses on the placement of distribution assignments. Before presenting this algorithm in detail, we remark that the computational complexity of PDCE and PRAE, and hence of DAP depends significantly on the number of iterations required for fully capturing the second-order effects of the elementary transformations. Hence, decoupling assignment patterns are a major means for enhancing the algorithm's efficiency. For distribution assignments this can be achieved by a simple *preprocess* which focuses on alignments and variables occurring in distribution specifications.

The Preprocess: Decoupling Distribution Assignment Patterns. Starting point of the preprocess are alignments and variables occurring in distribution specifications, which can be decoupled as follows:

- *Alignments*: Alignments are replaced by explicit distribution specifications where possible. For instance, REALIGN B(:) WITH A(:) is replaced by REDISTRIBUTE B(BLOCK), if *reaching distribution* analysis yields that array A is distributed by BLOCK.
- *Variables in distribution specifications*: Variables used in distribution specifications like variable i in BLOCK(i) or CYCLIC(i) may prevent distribution assignment motion. *Constant propagation* helps to suspend such blockades by replacing e.g. CYCLIC(i+j) by CYCLIC(3+1) and finally by CYCLIC(4) by expression folding.

Note that the preprocess for decoupling assignment patterns is not a prerequisite of our approach, rather a means of improving efficiency. This is important because there is of course a trade-off between the costs of the preprocess and the number of iterations subsequently saved. However, reaching distribution and constant propagation analysis, on which it relies, are usually exploited for other optimizations as well, and are thus in part for free.

Next we present the algorithm of *Pure* DAP in detail. It is the iterated sequential composition of the algorithms for PDCE and PRAE applied to the distribution assignment patterns of $\mathcal{D}$:

$$\mathsf{DAP}_{pure} \equiv (\mathsf{PDCE}_{\mathcal{D}}\ \ \mathsf{PRAE}_{\mathcal{D}})^{+}.$$

The iteration stops when both elimination steps with an interleaved assignment motion step fail to remove some code. Note, if the preprocess succeeds in decoupling all patterns of $\mathcal{D}$, $\mathsf{PDCE}_{\mathcal{D}}$ and $\mathsf{PRAE}_{\mathcal{D}}$ can equivalently be replaced by the more efficient algorithms specified by $(\mathsf{AS}_{\mathcal{D}}\ \ \mathsf{DCE}_{\mathcal{D}})$ and $(\mathsf{AH}_{\mathcal{D}}\ \ \mathsf{RAE}_{\mathcal{D}})$, respectively.

5.5 **Full** DAP

The Pure DAP-algorithm focuses on distribution assignment patterns. Thus, it does not capture second-order effects due to ordinary assignments. As a consequence the elimination of partially dead and partially redundant distribution assignments can get stuck by not resolving interdependences with ordinary assignments (cf. Figure 5(b), where the distribution assignment $\mathsf{dist(A)} := \mathsf{CYCLIC(k)}$ is blocked by the ordinary assignment k at node 4). The Full DAP-algorithm thus considers all assignment patterns of $\mathcal{P}$: it is the iterated sequential composition of the algorithms for PDCE and PRAE applied to all assignment patterns of $\mathcal{P}$:

$$\mathsf{DAP}_{full} \equiv (\mathsf{PDCE}\ \ \mathsf{PRAE})^{+}.$$

Figure 5 illustrates the different power of the algorithms of Pure DAP and Full DAP by means of the example of Figure 2. Whereas in this example a partially dead distribution assignment remains in the program after Pure DAP (cf. Figure 5(b)), the algorithm of Full DAP succeeds in eliminating all partially dead and partially redundant distribution assignments. Note, however, that also the transformationally weaker algorithm of Pure DAP succeeds here in removing all distribution assignments from the loop.

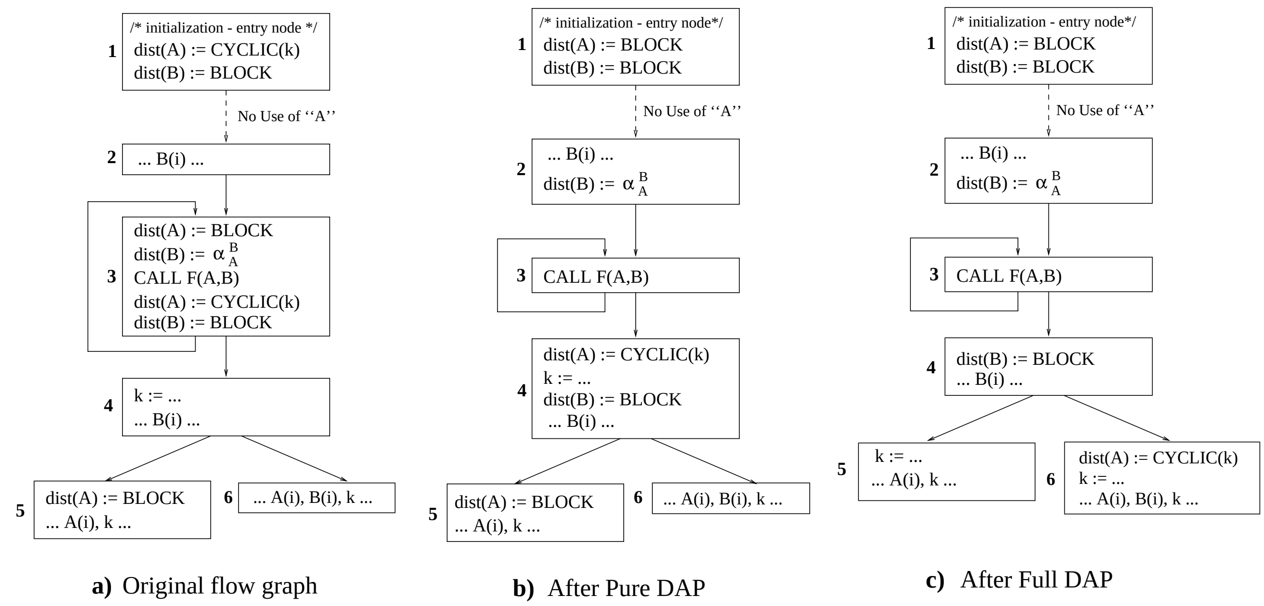


Figure 5: Pure DAP vs. Full DAP

5.6 Customized DAP-Variants: A Hierarchy of DAP-Algorithms

The algorithms of Pure DAP and Full DAP constitute the kernel of a hierarchy of DAP-algorithms of varying power and efficiency allowing customized DAP-variants according to a user's requirements or preferences. In this section we show how to selectively enhance efficiency and power.

Enhancing the power. Given a DAP-algorithm, its transformational power can easily be enhanced by replacing for ordinary assignments the partial dead-code elimination procedure by the more powerful *partial faint-code elimination* (PFCE) procedure (see [18] for details). In contrast to PDCE, PFCE eliminates assignments as "useless", where on each program continuation every right-hand side occurrence of its left-hand side variable is preceded by a modification of it, or occurs in an assignment whose left-hand side variable is "faint" as well. A simple example of a faint, however, not dead statement is an assignment like $x := x + 1$ occurring in a loop without another occurrence of x elsewhere in the program. For distribution patterns the notions of "dead" and "faint" coincide due to the syntactic limitations applying to their right-hand side terms.

Enhancing the efficiency. As pointed out, Pure DAP focuses on distribution patterns. In essence, the ratio underlying this choice 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 DAP, however, much more efficiently. In addition to limiting the statement patterns considered, this suggests to also limit the number of iterations performed. The extreme variant is here the 1-step heuristic focusing on distribution patterns:

$$\mathsf{DAP}^{1-step}_{pure} \equiv (\mathsf{AS}_{\mathcal{D}} \;\; \mathsf{DCE}_{\mathcal{D}}) \;\; (\mathsf{AH}_{\mathcal{D}} \;\; \mathsf{RAE}_{\mathcal{D}})$$

Note that $\mathsf{DAP}^{1-step}_{pure}$, which applys the elementary transformations only once, fails in capturing any second-order effects. No formal optimality criterion can be proven for it. However, it is extremely efficient and simultaneously it can be quite effective in practice. Of course, only empirical evidence for this conjecture could be given. Note, however, that in Figure 1 of Section 2 even the 1-step heuristic yields the optimal result, and in Figure 2 it succeeds in removing all distribution assignments from the loop.

Masking Distribution Assignments. Every DAP-algorithm of our hierarchy can be combined with *distribution assignment masking.* This means to prefix distribution assignments by masks ensuring that they are only performed if the actual distribution differs from the distribution which would be assigned to by the statement under consideration. Technically, $\mathsf{dist(A)} := \delta$ is replaced by: $\mathsf{if}\;\;(\mathsf{dist(A)} \neq \delta)\;\mathsf{dist(A)} := \delta$. Evaluating a mask consists of determining δ and comparing it with the actual distribution of array A. The effort required depends on the concrete implementation of the run-time system and may differ significantly for regular distributions like BLOCK and irregular distributions with mapping arrays, but usually it is much cheaper than the distribution statement it guards. Thus, masking can be used to reduce the overhead caused by partially redundant distribution assignments remaining in a program after DAP. We remark, however, that masking introduces overhead for program executions for which the masked occurrence is not redundant. Note that masking does not apply to remaining partially dead assignments: they have to be executed in any case.

5.7 Main Results: Optimality

Let G_{pure} and G_{full} be the programs resulting from our algorithms DAP_{pure} and DAP_{full}, respectively. Denoting additionally the sets of programs derivable from G by applying the four elementary transformations of Pure DAP and Full DAP in any order by $\mathcal{G}_{mix_\mathcal{D}} =_{df} \{G' \mid G \vdash^*_{(\mathsf{AS}_\mathcal{D} + \mathsf{DCE}_\mathcal{D} + \mathsf{AH}_\mathcal{D} + \mathsf{RAE}_\mathcal{D})} G'\}$ and by $\mathcal{G}_{mix} =_{df} \{G' \mid G \vdash^*_{(\mathsf{AS}+\mathsf{DCE}+\mathsf{AH}+\mathsf{RAE})} G'\}$, we have:

Theorem 1 (1st Optimality Theorem)
G_{full} and G_{pure} are locally best in $\mathcal{G}_{mix}$ and $\mathcal{G}_{mix_\mathcal{D}}$, respectively, i.e., they cannot be improved any further by means of admissible assignment sinkings (hoistings) or dead (redundant) assignment eliminations.

Actually, this is the best we can expect for G_{full} (G_{pure}) in $\mathcal{G}_{mix}$ ($\mathcal{G}_{mix_\mathcal{D}}$) in general. As illustrated in Section 5.3, $\mathcal{G}_{mix}$ ($\mathcal{G}_{mix_\mathcal{D}}$) lacks in general the existence of a program being "globally best", but provides a number of programs being "locally best", i.e., which cannot be improved any further by means of the component transformations of PDCE and PRAE. In particular, any remaining partially dead or partially redundant assignment in G_{full} (G_{pure}) cannot be removed by the respective class of assignment sinkings/hoistings and dead/redundant assignment eliminations under consideration without modifying the branching structure of the program or impairing some program executions.

In practice, even the simple sequential composition of PDCE and PRAE without iterating (i.e., $\mathsf{DAP}^{smpl} \equiv (\mathsf{PDCE}\ \ \mathsf{PRAE}))$ which results in G^{smpl}_{pure} and G^{smpl}_{full}, respectively, often succeeds in completely removing partially dead and partially redundant assignments. Though this does not hold in general (in particular, G^{smpl}_{pure} and G^{smpl}_{full} need not to be locally best in $\mathcal{G}_{mix_\mathcal{D}}$ and $\mathcal{G}_{mix}$, respectively), the following optimality result applies to them. Let $\mathcal{G}_{pure} =_{df} \{G' \mid G \vdash^*_{(\mathsf{AS}_\mathcal{D}+\mathsf{DCE}_\mathcal{D})} \bar{G} \vdash^*_{(\mathsf{AH}_\mathcal{D}+\mathsf{RAE}_\mathcal{D})} G'\}$ and $\mathcal{G}_{full} =_{df} \{G' \mid G \vdash^*_{(\mathsf{AS}+\mathsf{DCE})} \bar{G} \vdash^*_{(\mathsf{AH}+\mathsf{RAE})} G'\}$, where $\bar{G}$ is assumed to be invariant under further assignment sinkings (up to local reorderings in basic blocks) and dead code eliminations. Then we have (cf. [18, 19]):

Theorem 2 (2nd Optimality Theorem)

G_{full}^{smpl} and G_{pure}^{smpl} are optimal in $\mathcal{G}_{full}$ and $\mathcal{G}_{pure}$, respectively.

6 Experimental Results

In this section we report on first practical experiences showing the performance improvements which can be achieved by DAP. Figure 6 shows the original and optimized version of an HPF code fragment consisting of a computational kernel accessing a prescriptively distributed array. Subroutine F1 sweeps over the columns calculating new values for X(i,j) by looking at the neighboring elements X(i-1,j) and X(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,*). To demonstrate the effectivity of DAP, we measured for the original and the optimized program version the execution time of the do-loop of the main program. For the optimized version, we added also the time required to restore the mapping of array A which is only necessary when the then-branch is executed, i.e., we made a worst-case assumption and took for the optimized version the maximum execution time. The results are shown in Figure 7. 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 7 illustrate the importance of this kind of optimization for varying array sizes and loop iterations. The dramatic improvement observed is not surprising in the light of average remapping times. Ramaswamy et al. [25] developed highly efficient remapping routines: remappings with different processor sets (from 8 to 16 processors) and redistributions from (cyclic(3),block) to (cyclic,cyclic(5)), (cyclic(3),cyclic(7)) to (cyclic(5),cyclic), and (cyclic,$*$) to ($*$,cyclic) took for arrays with sizes 100×100, 200×200, and 400×400 on the average about 5 msec, 10 msec, and 32 msec on an Intel Paragon, respectively.

7 Conclusions

Eliminating *partially dead* and *partially redundant* redistributions is of key importance to gain efficiency. Based on the recently developed algorithms for PDCE and PRAE of [18] and [19] working for standard sequential programs, we showed how to adapt and combine them to optimize data remappings in data-parallel languages. Second-order effects between PDCE and PRAE showing up by combining them required not only a refined optimality investigation, but also led to a hierarchy of algorithms for distribution assignment placement of varying power and efficiency offering cus-

(a) Basic distribution assignment generation.

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

```
! internal subprogram
subroutine F1(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 DAP.

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

```
! internal subprogram
subroutine F1(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 6: A basic and an optimized program version of an HPF code fragment.

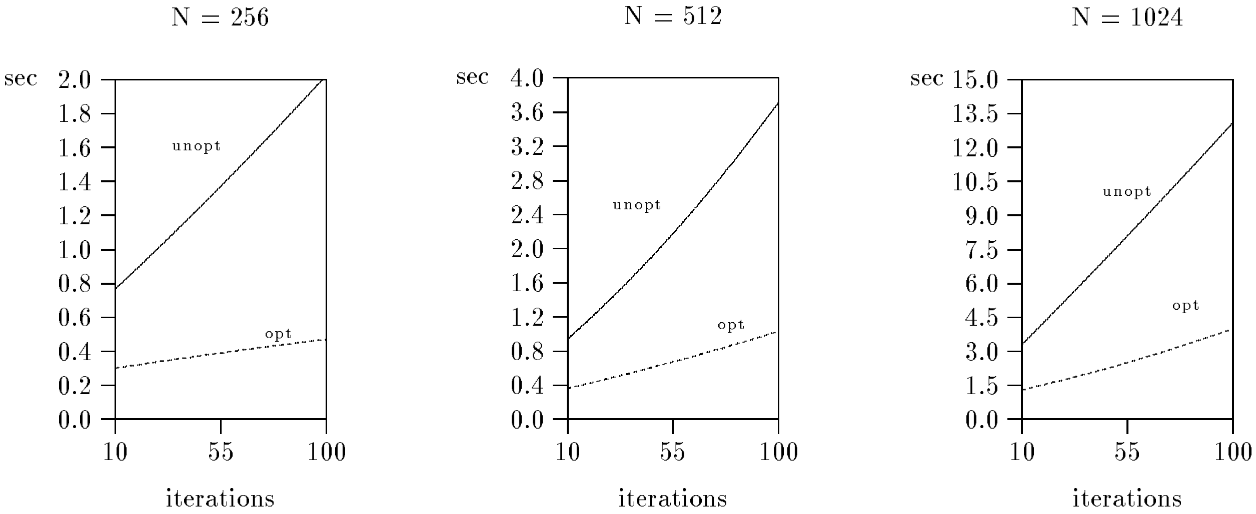


Figure 7: Measured run-times of Example of Fig. 6 on a Meiko CS-2 with 16 processors.

tomized solutions according to a user's requirements or preferences. This ranges from extremely efficient one-step heuristics to extremely powerful procedures like the enhanced Full DAP-algorithm resolving uniformly all second-order effects between all assignment patterns (ordinary and distribution assignments). Moreover, our approach is not limited to distributed-memory systems. In fact, data locality is also an important issue for non-uniform memory access (NUMA) architectures like virtual shared memory multiprocessors. In addition, we are currently investigating an interprocedural extension of our approach (cf. [14]) based on [20], and how it compares to both its intraprocedural counterpart and other interprocedural algorithms. An implementation of the complete approach within the VFCS system [2] is in progress.

Acknowledgements: We are very grateful to Shankar Ramaswamy and Prithviraj Banerjee for making their HPF redistribution library available to us.

References

[1] G. Agrawal, J. Saltz, and R. Das. Interprocedural partial redundancy elimination and its application to distributed memory compilation. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation* (*PLDI'95*) (*La Jolla, CA*), volume *30*,6 of *ACM SIGPLAN Notices*, pages 258–269, 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, February 1996.

[3] F. Coelho and C. Ancourt. Optimal compilation of HPF remappings. *Journal of Parallel and Distributed Computing*, 38(2):229–236, November 1996.

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

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

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

[7] High Performance Fortran Forum. High Performance Fortran language specification version 2.0. Technical report, Rice University, Houston,TX, January 1997. Available via HPFF home page: http://www.crpc.rice.edu/HPFF.

[8] 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.

[9] A. Geser, J. Knoop, G. Lüttgen, O. Rüthing, and B. Steffen. Non-monotone fixpoint iterations to resolve second order effects. In *Proceedings of the 6th International Conference on Compiler Construction (CC'96) (Linköping, Sweden)*, Lecture Notes in Computer Science, vol. 1060, pages 106 – 120. Springer-Verlag, Heidelberg, Germany, 1996.

[10] 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.

[11] S.K.S. Gupta, C.-H. Huang, P. Sadayappan, and R. W. Johnson. Implementing Fast Fourier Transforms on Distributed-Memory Multiprocessors Using Data Redistributions. *Parallel Processing Letters*, 4(4):477–488, November 1994.

[12] 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*, pages 522–534, Minneapolis, MN, November 1992.

[13] 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.

[14] J. Knoop and E. Mehofer. Interprocedural distribution assignment placement: More than just enhancing intraprocedural placing techniques. In *Proceedings of the 5th IEEE International Conference on Parallel Architectures and Compilation Techniques* (*PACT'97*) (*San Francisco, California*), pages 26 – 37. IEEE Computer Society, Los Alamitos, 1997.

[15] J. Knoop and E. Mehofer. Optimal distribution assignment placement. In *Proceedings of the 3rd European Conference on Parallel Processing* (*Euro-Par'97*) (*Passau, Germany*), Lecture Notes in Computer Science, vol. 1300, pages 364 – 373. Springer-Verlag, Heidelberg, Germany, 1997.

[16] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation* (*PLDI'92*) (*San Francisco, CA*), volume *27*,7 of *ACM SIGPLAN Notices*, pages 224 – 234, 1992.

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

[18] 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*) (*Orlando, FL*), volume *29*,6 of *ACM SIGPLAN Notices*, pages 147–158, 1994.

[19] 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*) (*La Jolla, CA*), volume *30*,6 of *ACM SIGPLAN Notices*, pages 233–245, 1995.

[20] 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.

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

[22] 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.

[23] D.J. Palermo, E.W. Hodges, and P. Banerjee. Interprocedural array redistribution data-flow analysis. In *Proceedings of the 9th Workshop on Languages and Compilers for Parallel Computing*, San Jose, CA, August 1996.

[24] L. Prylli and B. Tourancheau. Efficient block-cyclic data redistribution. In *Proceedings of the 2rd European Conference on Parallel Processing (Euro-Par'96) (Lyon, France)*, Lecture Notes in Computer Science, vol. 1123, pages 155 – 164. Springer-Verlag, Heidelberg, Germany, 1996.

[25] S. Ramaswamy, B. Simons, and P. Banerjee. Optimizations for efficient array redistribution on distributed memory multicomputers. *Journal of Parallel and Distributed Computing*, 38(2):217–228, November 1996.

[26] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Conference Record of the 15th ACM Symposium on Principles of Programming Languages (POPL'88) (San Diego, CA)*, pages 2 – 27, 1988.

[27] 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.

A Hoistability and Redundant Assignment Analysis

Assignment Hoisting Analysis (AH): (Let $\alpha \in \mathcal{P}$)

- $\mathbf{LocHoist}_n(\alpha)$: There is a hoisting candidate of α in n.
- $\mathbf{LocBlock}_n(\alpha)$: The hoisting of α is blocked by some instruction of n.

$$\begin{array}{lcl} \mathbf{N\text{-}HOIST}_n & = & \mathbf{LocHoist}_n + \mathbf{X\text{-}HOIST}_n * \overline{\mathbf{LocBlock}_n} \\ \mathbf{X\text{-}HOIST}_n & = & \begin{cases} \mathit{false} & \mathit{if} \;\; n = \mathbf{e} \\ \prod\limits_{m \in succ(n)} \mathbf{N\text{-}HOIST}_m & \mathit{otherwise} \end{cases} \end{array}$$

Insertion Points:[c] [d]

$$\begin{array}{lcl} \mathbf{N\text{-}INSERT}_n & =_{df} & \mathbf{N\text{-}HOIST}^\star_n * \begin{cases} \mathit{true} & \mathit{if}\; n = \mathbf{s} \\ \sum\limits_{m \in pred(n)} \overline{\mathbf{X\text{-}HOIST}^\star_m} & \mathit{otherwise} \end{cases} \\ \mathbf{X\text{-}INSERT}_n & =_{df} & \mathbf{X\text{-}HOIST}^\star_n * \mathbf{LocBlock}_n \end{array}$$

Redundant Assignment Analysis (RAE):[e] (Let $\alpha \in \mathcal{P}$ such that the left-hand side of α does not occur in its right-hand side term. Let s denote the first instruction of $\mathbf{s}$.)

- $\mathbf{IsOcc}_\iota(\alpha)$: Instruction ι is an assignment of pattern α.
- $\mathbf{AssTransp}_\iota(\alpha)$: Instruction ι is transparent for α, i.e., neither α's left-hand side nor any of the arguments needed in α are modified by ι (note that $\iota \equiv A := ...$ is transparent for $\alpha \equiv dist(A) := ...$).

$$\begin{array}{lcl} \mathbf{N\text{-}RED}_\iota & = & \begin{cases} \mathit{false} & \mathit{if} \;\; \iota = s \\ \prod\limits_{\hat{\iota} \in pred(\iota)} \mathbf{X\text{-}RED}_{\hat{\iota}} & \mathit{otherwise} \end{cases} \\ \mathbf{X\text{-}RED}_\iota & = & \mathbf{IsOcc}_\iota + \mathbf{AssTransp}_\iota * \mathbf{N\text{-}RED}_\iota \end{array}$$

Elimination Points:[c] $\mathbf{ELIM}_\iota =_{df} \mathbf{N\text{-}RED}^\star_\iota$

Table 2: Assignment Hoisting and Redundant Assignment Analysis

[c] $\mathbf{N\text{-}HOIST}^\star$, $\mathbf{X\text{-}HOIST}^\star$, and $\mathbf{N\text{-}RED}^\star$ denote the greatest solution of the equation systems for hoistability and redundancy, respectively.

[d] This means to replace all hoisting candidates of the assignment pattern under consideration by instances of this pattern inserted at program points satisfying **N-INSERT** or **X-INSERT**.

[e] The analysis is employed at the instruction level, which eases presentation. It can straightforward be modified to work on basic blocks instead.

B Pragmatics

In this section we discuss a variety of pragmatic aspects related to our approach.

Uniform Integration of Output Statements and Conditions. In practice, also second-order effects due to output statements and conditions must be taken into account. Actually, it is straightforward to enhance our approach accordingly. Considering *output statements* like assignments without left-hand sides suffices for handling them uniformly in our approach. Similarly, this can be achieved for *conditions*. Replace each condition by an assignment to a fresh variable followed by the branch statement with the original condition replaced by the fresh variable. Obviously, the fresh variable does not block any assignment just as it did not occur in the original program. On the other hand, the statement added can be subjected to PDCE and PRAE like any other statement. Moreover, it is pragmatically advantageous here to associate lexically identical conditions with the same fresh variable in order to open opportunities for redundancy eliminations.

Implementation Aspects. Conceptually, PDCE (PRAE) consists of two separate steps: AS (AH) and DCE (RAE). This simplifies the presentation of the algorithms. In the implementation, however, both steps can be combined, i.e., the effect of the motion step has not to be made explicit.

C Critical Edges

Optimizations based on code motion like partial redundancy elimination, partial dead-code elimination or partially redundant assignment elimination can be blocked by *critical edges* in a flow graph, i.e., by edges leading from a node with more than one successor to a node with more than one predecessor (cf. [4, 5, 16, 17]). In Figure 8 this is illustrated using partial dead-code elimination as example.

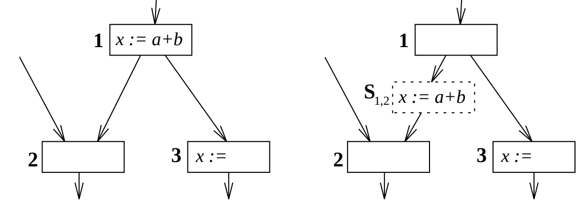


Figure 8: Critical Edges

In Figure 8(a) the assignment $x := a + b$ at node **1** is partially dead with respect to the assignment at node **3**. However, it cannot safely be eliminated by moving it to its successors, because this may introduce a new assignment on a path entering node **2** on the left branch. On the other hand, it can safely be eliminated after inserting a synthetic node $\mathbf{S}_{1,2}$ in the critical edge $(\mathbf{1}, \mathbf{2})$, as illustrated in Figure 8(b). Dually, this holds for partially redundant assignment elimination. In the present article we therefore restrict our attention to programs where every critical edge has been split by inserting a synthetic node.

D Iterated Second-Order Effects of PDCE and PRAE

In this section we illustrate second-order effects of PDCE and PRAE considering the example of Figure 9. The first application of PDCE eliminates dist(A) := CYCLIC(i) at node **6**, which turns the assignment dist(A) := BLOCK(i) at node **7** to be redundant and, hence, is removed by the subsequent PRAE-step, which eliminates also a second distribution assignment on the rightmost branch at node **27** as shown in Figure 11. These eliminations suspend the blockades of the partially dead occurences of dist(B) := $\alpha_{\mathsf{A}}^{\mathsf{B}}$ at nodes **12** and **26**, which can then be eliminated by a second application of PDCE resulting in Figure 12. As a consequence, dist(B) := CYCLIC(j) at node **32** becomes totally redundant, which is eliminated by the subsequent PRAE-step. This suspends the blockade of the partially dead assignment to variable z at node **31**, which can now be eliminated by another PDCE-step yielding the program of Figure 14 which is stable under further admissible assignment motions and dead (redundant) assignment eliminations.

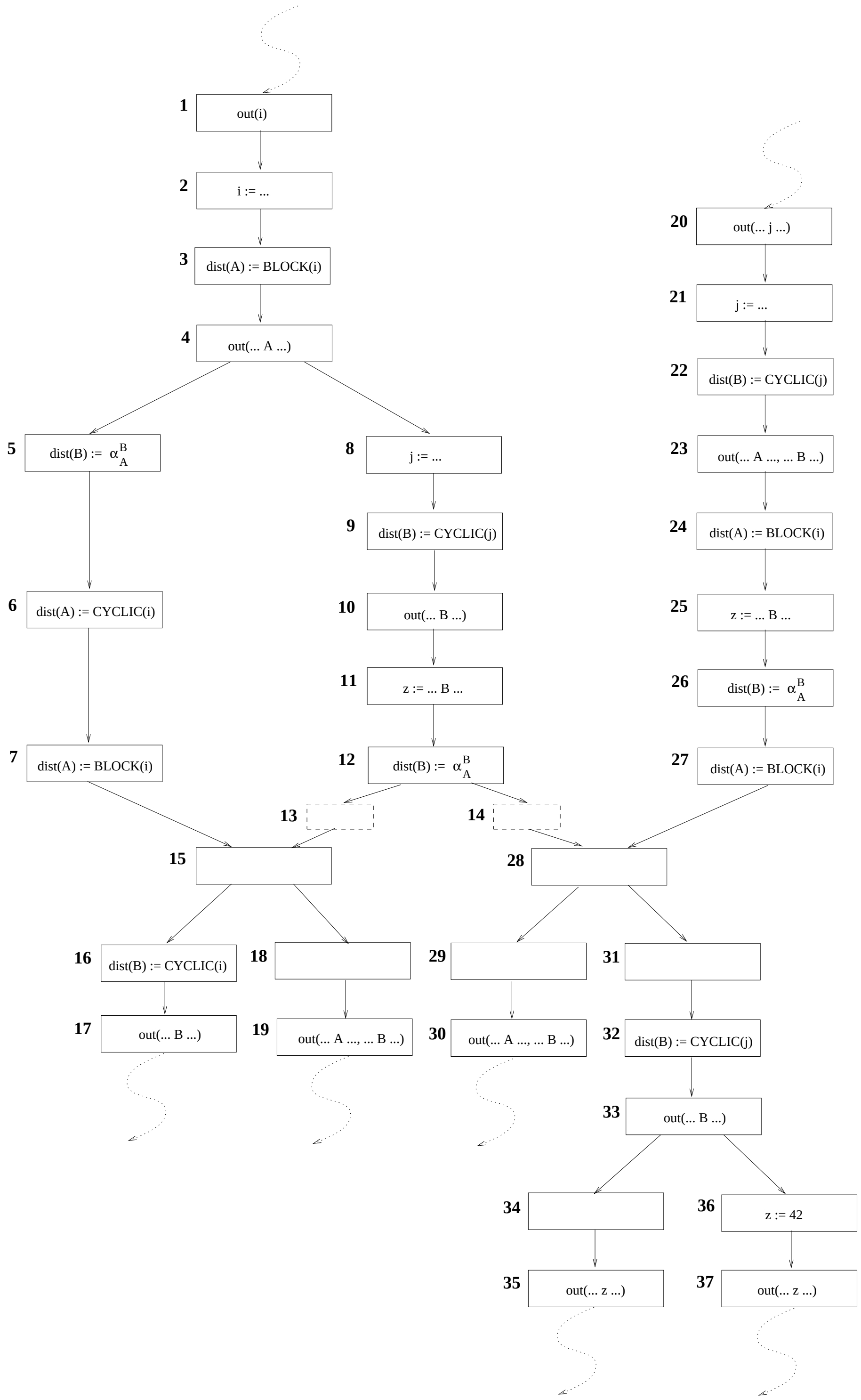


Figure 9: The Original Flow Graph

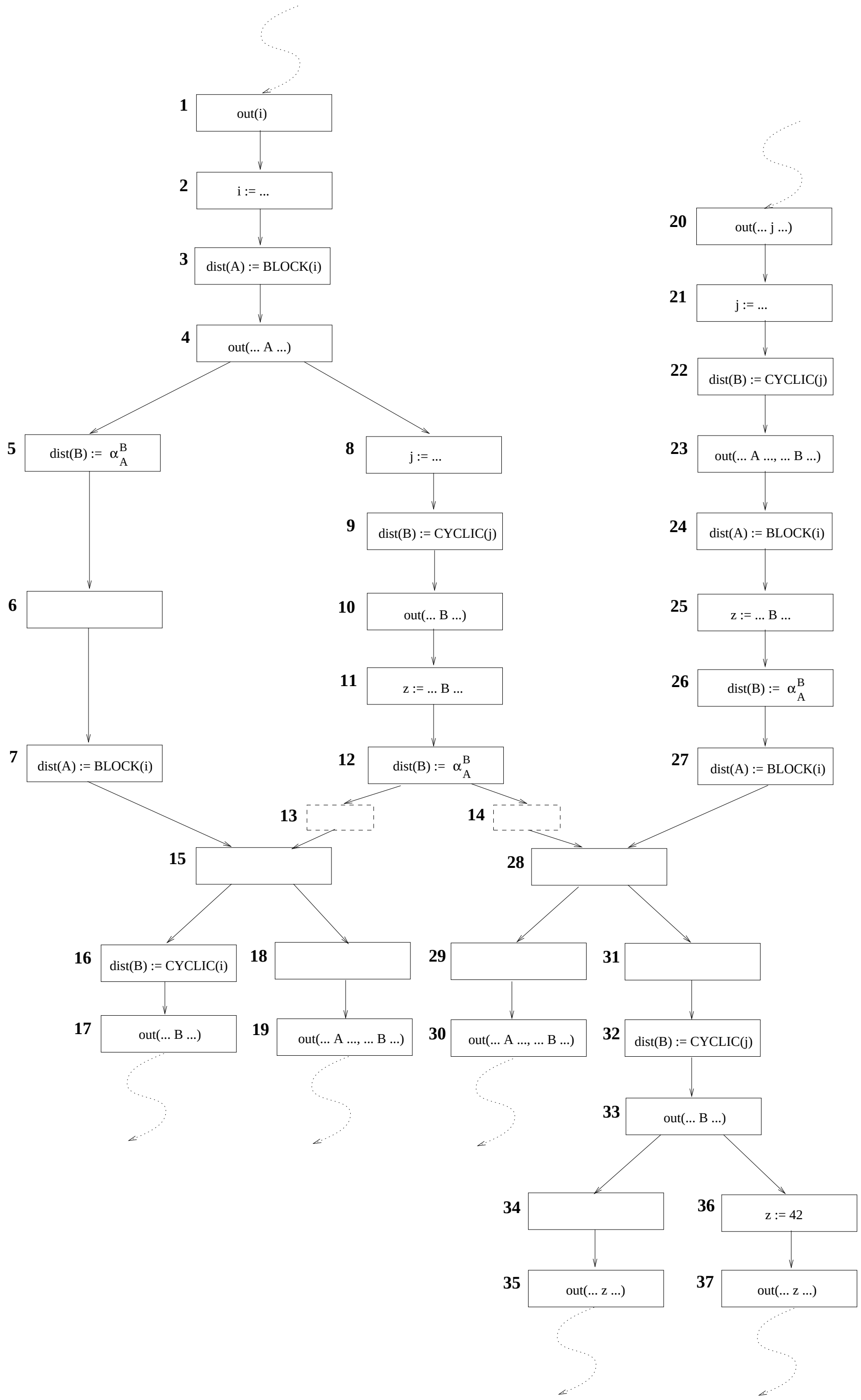


Figure 10: After PDCE

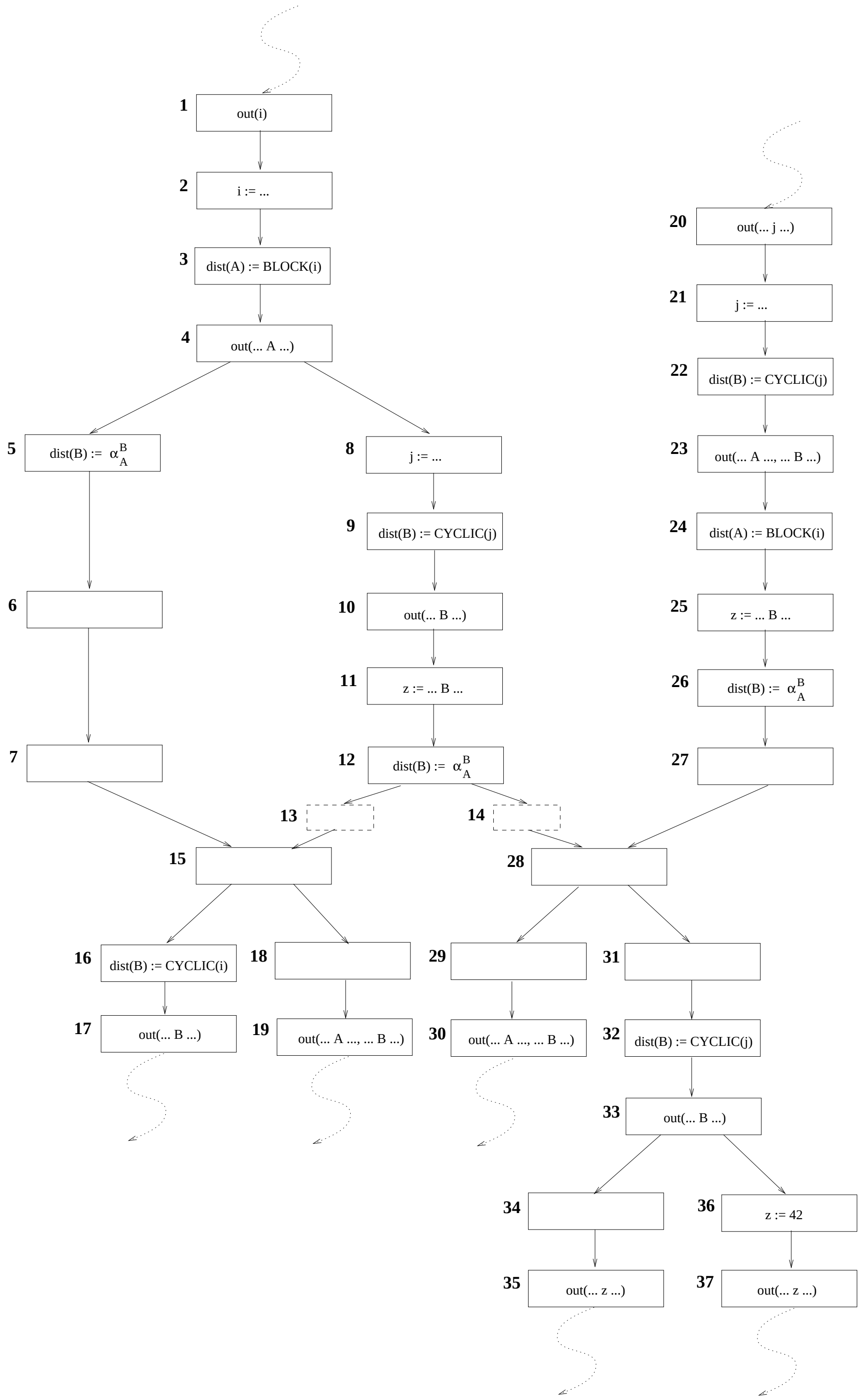


Figure 11: After PDCE - PRAE

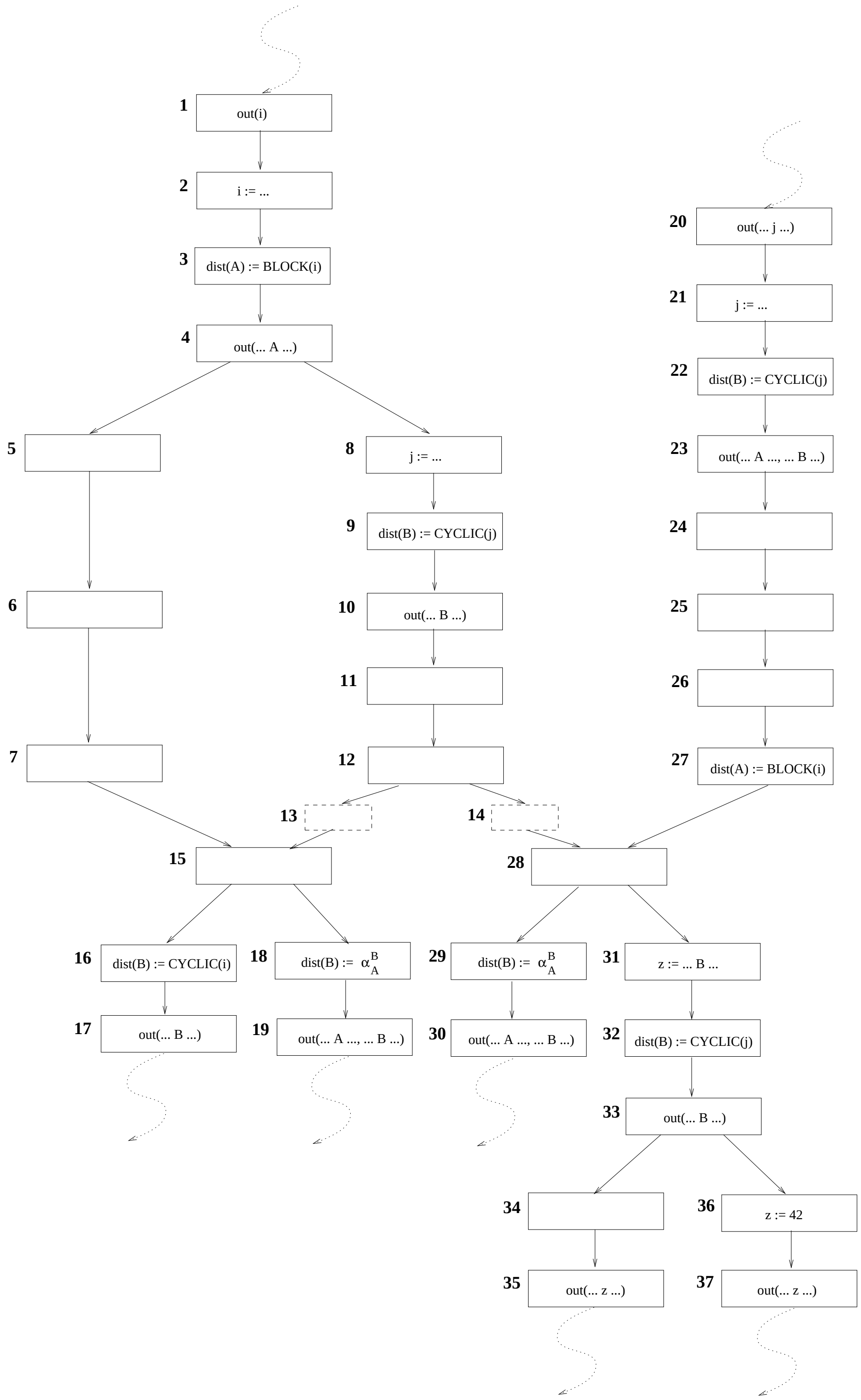


Figure 12: After PDCE - PRAE - PDCE

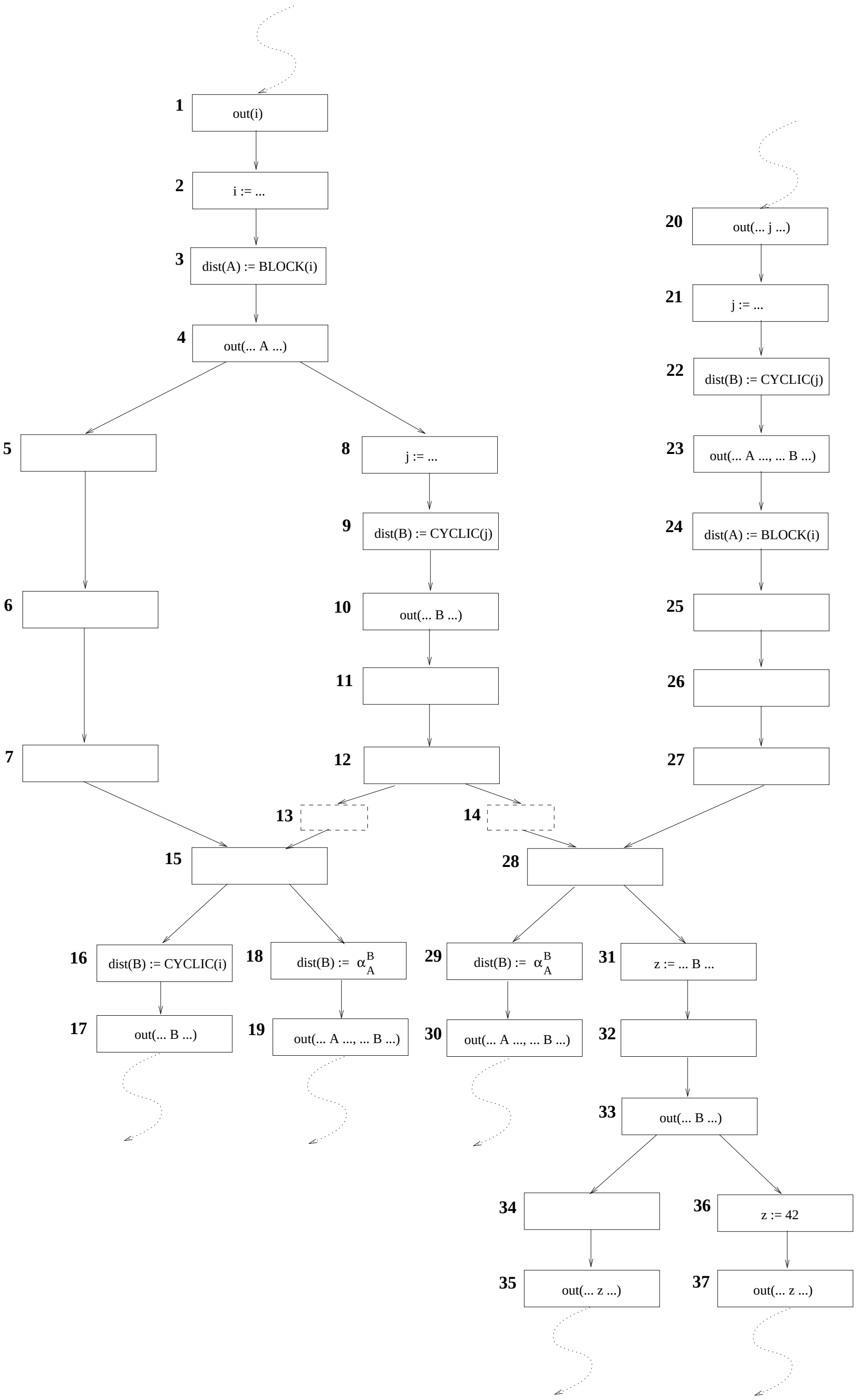


Figure 13: After PDCE - PRAE - PDCE - PRAE

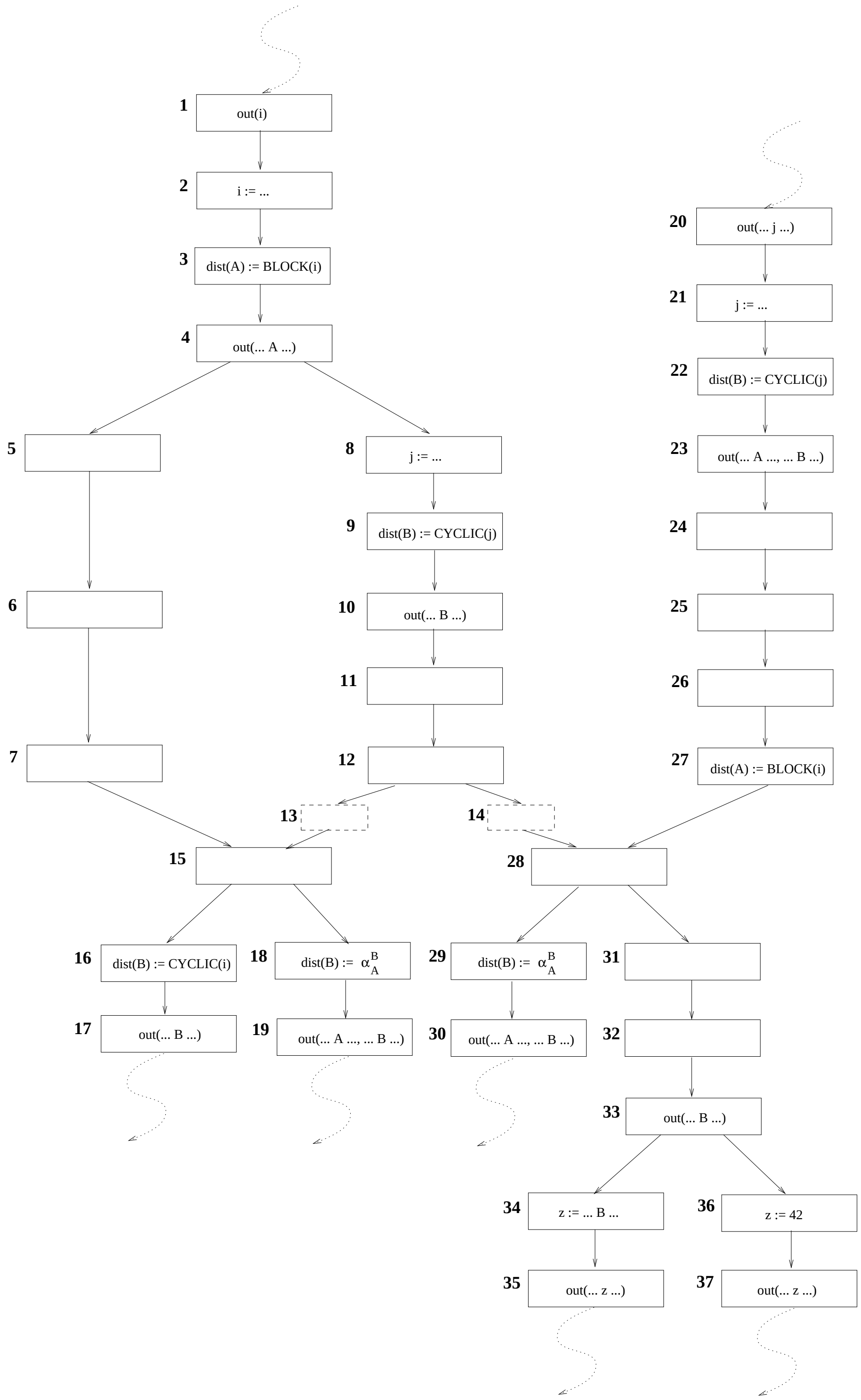


Figure 14: Stable after PDCE - PRAE - PDCE - PRAE - PDCE