Optimal Distribution Assignment Placement

Jens Knoop[1] and Eduard Mehofer[2]

[1] Universität Passau, Fakultät für Mathematik und Informatik, Innstraße 33, D-94032 Passau, Germany (e-mail: knoop@fmi.uni-passau.de)
[2] Universität Wien, Institut für Softwaretechnik und Parallele Systeme, Liechtensteinstr. 22, A-1090 Vienna, Austria (e-mail: mehofer@par.univie.ac.at)

Abstract. Dynamic data redistribution is a key technique for maintaining data locality and workload balance in data-parallel languages like HPF. On the other hand, redistributions can be very expensive and significantly degrade a program's performance. In this article, we present a novel and aggressive approach for avoiding unnecessary remappings by eliminating *partially dead* and *partially redundant* distribution changes. Basically, this approach evolves from extending and combining two algorithms for these optimizations achieving optimal results for sequential programs. Optimality, however, becomes more intricate by the combination. Unlike the sequential setting the data-parallel setting leads to a hierarchy of algorithms of varying power and efficiency fitting a user's individual needs. The power and flexibility of the new approach are demonstrated by illustrating examples. First practical experiences underline its importance and effectivity.

1 Motivation

The user-controlled distribution of data across the local memories of the processing nodes is a central feature of data-parallel languages like *High Performance Fortran* (HPF) [3], *Fortran D* [4], or *Vienna Fortran* [13]. A program's performance can critically depend on the distribution chosen. Dynamic data redistributions, e.g., in case of varying computational kernels or dynamically varying processor workloads, are thus a major means for improving the performance. On the other hand, remappings can be quite expensive as communication is required to migrate the array elements to their new owning processors. Unnecessary distribution changes can therefore significantly degrade a program's performance. Avoiding them is of key importance to gain efficiency.

In this article we present a novel and 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 achieving optimal results for sequential programs (cf. [7,8]). 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 captures removal of unnecessary remappings uniformly 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 can be overcome as usual by repeated applications of the elementary transformations, the interleaving of all four elementary transformations reveals additional interdependences of the transformations themselves making optimality more intricate as the result depends on the particular sequence of the elementary transformations. 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 efficiency fitting a user's individual needs. While the basic version of *Pure* DAP focusses on distribution assignments and resolves second-order effects among them, the algorithm of *Full* DAP resolves even all second-order effects between ordinary and distribution assignments, and achieves results, which cannot be improved any further by means of the elementary transformations. Partially dead and partially redundant distribution assignments remaining in the program cannot be eliminated further without changing its branching structure or impairing some of its executions.

2 Illustrating Example and Practical Experiences

Since HPF allows to change the distribution of arrays explicitly as well as implicitly with *redistribute/realign directives* and at subprogram boundaries, respectively, we introduce as in [10] *distribution assignments*, which are part of the intermediate program representation and allow a uniform treatment of all remappings. Hence, distribution assignments are inserted by the compiler whenever an array is associated with a distribution.

The example of Figure 1 illustrates the essential features and 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 different distribution assignments. Parameters in distribution specifications like in BLOCK(i) and CYCLIC(i) introduce dependences between ordinary and distribution assignments, as well. Both kinds of dependences may result in second-order effects as illustrated in Figure 1(a) and 1(b).

The distribution assignments in node **3** just before and just after subroutine call F are necessary to establish the requested distributions on entry and to restore the original one on exit according to the HPF semantics. The alignment of array B to array A is denoted by $\alpha_{\mathsf{A}}^{\mathsf{B}}$. Note that none of the distribution assignments within the loop is (totally) redundant or dead. However, 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. 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. Sinking of the ordinary assignment k suspends this blockade and subsequently allows the elimination of the partially dead assignment dist(A) := CYCLIC(k), which, as a second-order effect, turns dist(A) := BLOCK at node **5** to be redundant and, hence, can be

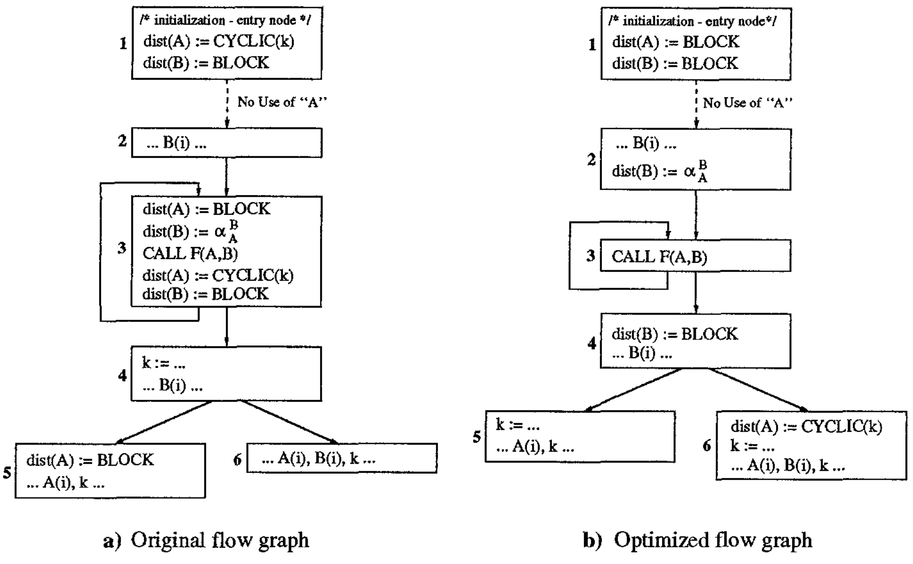

Fig. 1. Motivating example: Illustrating second-order effects and the power of DAP.

eliminated. 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 also suspends the blockade of dist(B):= α_A^B at the entry of the loop, which, as a further second-order effect, can now be moved to node **2**.

Figure 1(b) 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.

Practical experiences: Reducing the number of remappings, in particular within loops, may decrease the runtime of programs tremendously (cf. performance results of [10]). Such performance improvements are not surprising in the light of average remapping costs. Ramaswamy et al. [12] 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.

Related work: Redistribution analysis has been addressed by several researchers. Hall et al. [5] 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. [2] describe an optimization which reduces the communication amount by removing useless remappings and taking advantage of replications to shorten individual remappings. Optimal 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 Ramaswamy et al. [12], whose focus lies on the automatic generation of efficient routines for migrating the array elements to their new owning processors. Palermo et al. [11] 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 is not considered.

3 Preliminaries

As usual we represent a program by a *directed flow graph* $G = (N, E, \mathrm{s}, \mathrm{e})$ with node set N and edge set E. Nodes $n \in N$ represent *basic blocks* of instructions, edges $(m, n) \in E$ the nondeterministic branching structure of G, and s and e the unique *start node* and *end node* of G, which are assumed to have no incoming and outgoing edges, respectively. All statements of a program are classified as follows: (ordinary) *assignment statements* including both scalar and indexed variables; the *empty statement* skip; *distribution assignments* of the form $dist(A) := \delta$, 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. Finally, all edges leading from a node with several successors to a node with several predecessors are assumed to be split by a synthetic node. This is typical for code motion transformations in order to avoid that the motion process gets stuck (cf. [7,8]).

4 Distribution Assignment Placement

In this section we stepwise develop our hierarchy of DAP-algorithms starting with the algorithms of *pure* and *full* DAP providing first 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. For convenience, we consider an arbitrary but fixed flow graph G. Moreover, let $\mathcal{P}$ denote the set of ordinary and distribution assignment patterns occurring in G, and let $\mathcal{D} \subseteq \mathcal{P}$ denote the subset of distribution patterns.

As illustrated in Section 2 the essence of DAP is to avoid unnecessary executions of distribution assignments at runtime. 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. Hence, 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 discussed in [7] and [8] presenting algorithms for ***partially dead code elimination*** (PDCE) and ***partially redundant assignment elimination*** (PRAE), respectively. Below, we are going to show how to enhance these algorithms being developed for a standard sequential program setting to the data-parallel setting, and how to combine them uniformly in order to arrive at a hierarchy of user-customized DAP-algorithms.

4.1 The Component Transformations of DAP: PDCE and PRAE

Like DAP, PDCE and PRAE consist conceptually of two elementary transformations each: *assignment sinkings* (AS) and *dead code eliminations* (DCE), and *assignment hoistings* (AH) and *redundant assignment eliminations* (RAE), respectively. Assignment sinkings (hoistings) move assignments as far as possible in the (opposite) direction of the control flow (i.e., while maintaining the program semantics). Intuitively, this places them in a context as *specific* (*general*) as possible, and maximizes the potential of dead (redundant) code, which subsequently is removed by dead (redundant) assignment elimination.

The analyses for DCE and RAE coincide with their classical counterparts. Thus, we only recall the intuition underlying the AS- and AH-analysis, and how to adapt them to the data-parallel setting considered here (a detailed presentation can be found in [6]). The point of these analyses is to restrict assignment sinkings and hoistings to ***admissible*** ones, i.e., those preserving the semantics. In essence, admissibility requires that assignments are never moved across instructions ***blocking*** them, 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 fact, the same constraints we impose on distribution assignments. As a consequence, the AS- and AH-analysis of [7,8] apply directly to distribution assignments as well. In particular, this guarantees that assignment sinkings (hoistings) respect the distribution proposed by the programmer: for each array reference the distributions in the original and the optimized program are identical.[1]

The PDCE- and PRAE-Algorithms and their Optimality. Following [7,8] 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 and PRAE until the program stabilizes. This is conveniently expressed by means of the following regular-expression like terms

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

[1] 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 blocking constraint for ordinary and distribution assignments differently.

where AS (AH), and DCE (RAE) denote a single application of the assignment sinking (hoisting), and dead (redundant) assignment elimination procedure to all assignment patterns of $\mathcal{P}$. As shown in [7,8] the programs resulting from PDCE or PRAE, respectively, are *optimal*: they are *best*, i.e., *better* than any other program in the set of programs $\mathcal{G}_{pdce} =_{df} \{G' \mid G \vdash^*_{(\mathsf{AS+DCE})} G'\}$ and $\mathcal{G}_{prae} =_{df} \{G' \mid G \vdash^*_{(\mathsf{AH+RAE})} G'\}$ derivable from G by means of sequences of admissible assignment sinkings (hoistings) and dead (redundant) assignment eliminations, where a program G' is *better* than a program G'' iff for every assignment pattern the number of assignments executed on each path in G' is less or equal to that in G''. In particular, the programs finally resulting from PDCE and PRAE do not depend on the specific order of the elementary transformations: they are uniquely determined up to (irrelevant) local reorderings in basic blocks (cf. [7,8]).

4.2 Interdependences between PDCE and PRAE

As recalled above, the elementary transformations of PDCE or PRAE can be applied in any order without affecting the program finally resulting (cf. [7,8]). In fact, there is a "globally best" program in the set of programs being derivable by admissible assignment sinkings (hoistings) and dead (redundant) assignment eliminations, which finally is reached by any 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 2. Applying PDCE first we arrive at the program of Figure 2(b), applying PRAE first we arrive at the program of Figure 2(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 2(c) contains precisely one distribution assignment, there is a path through the fragment of Figure 2(b) being free of distribution assignments, and another one containing two.

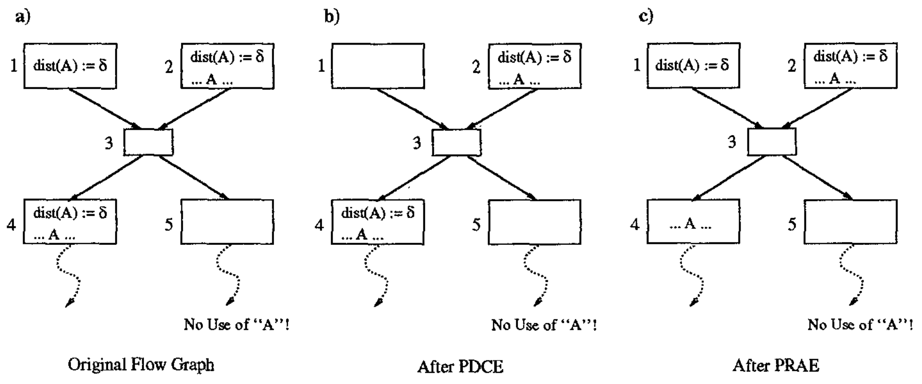

Fig. 2. Interdependences between PDCE and PRAE.

As pointed out by this example, PDCE and PRAE influence each other by mutually removing opportunities for their respective counterpart. Nonetheless, the process of interleaving all four elementary transformations 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". However, different transformation sequences will usually come up with different locally best programs, which, in general, behave better on some program paths, but worse on others. Therefore, we concentrate here on the question in which order to apply the elementary transformations to achieve fast stabilization. Obviously, AS- and AH-steps should always be interleaved with DCE- and RAE-steps (as AS and AH just 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 [6] for an 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? Though in general any decision at this point is arbitrary, two reasons suggest starting with PDCE. First, the situation displayed in the example of Figure 1, where PDCE is a prerequisite for enabling PRAE, can be considered quite typical for data-parallel programs. Second, PRAE never creates partially dead assignments, but PDCE may create partially redundant ones. Though not sufficient, this enlarges the chance that PDCE followed by PRAE already terminates with a locally best program as demonstrated in Section 2.

4.3 Pure DAP

Pure DAP focusses 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 focusses on alignments and variables occurring in distribution specifications. It replaces *alignments* 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* 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) with CYCLIC(3+1) and finally by CYCLIC(4) by expression folding. We remark that there is 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 also exploited for other optimizations, and are thus in part for free.

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

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

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.

4.4 Full DAP

The Pure DAP-algorithm focusses 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 redundant distribution assignments can get stuck by not considering the interdependences with ordinary assignments (cf. Figure 1 where distribution assignment dist(A) := 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 1 illustrates the power of the Full DAP-algorithm, where it is unique to eliminate all partially dead and partially redundant distribution assignments.

4.5 Customized DAP-Variants: Enhancing Power and Efficiency

The algorithms of Pure 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. *Efficiency* for example, can simply be enhanced by limiting the number of iterations of the component transformations. The ratio underlying this heuristic to yield algorithms being still reasonably effective is that distribution assignments are used in a quite restricted manner in practice only. The extreme variant is here the one-step heuristic focussing on distribution assignments: $\mathsf{DAP}_{pure}^{one-step} \equiv (\mathsf{AS}_{\mathcal{D}} \;\; \mathsf{DCE}_{\mathcal{D}}) \;\; (\mathsf{AH}_{\mathcal{D}} \;\; \mathsf{RAE}_{\mathcal{D}})$. On the other hand, the transformational *power* of the DAP-algorithms can easily be enhanced by replacing the partial dead-code elimination procedure by the more powerful *partial faint-code elimination* (PFCE) procedure (see [7] for details). Finally, all DAP-algorithms can be combined with *distribution assignment masking.* This ensures that distribution assignments (at the price of a much cheaper runtime test) are executed at runtime only if they have a non-trivial effect.

4.6 Main Results: Optimality

Let G_{pure} and G_{full} be the programs resulting from our algorithms DAP_{pure} and DAP_{full}, respectively. Denoting the sets of programs derivable from G by applying the four elementary transformations of pure 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, analogously, 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 almost the best we can expect for G_{full} (G_{pure}) in $\mathcal{G}_{mix}$ ($\mathcal{G}_{mix_\mathcal{D}}$). As illustrated in Section 4.2, $\mathcal{G}_{mix}$ ($\mathcal{G}_{mix_\mathcal{D}}$) 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 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. [7,8]):

Theorem 2 (2nd Optimality Theorem).
G^{smpl}_{full} and G^{smpl}_{pure} are optimal in $\mathcal{G}_{full}$ and $\mathcal{G}_{pure}$, respectively.

5 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 [7] and [8] 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 providing customized solutions fitting a user's individual needs. This ranges from extremely efficient one-step heuristics to extremely powerful procedures resolving all second-order effects between different assignment patterns like the enhanced Full DAP-Algorithm. Currently, we are investigating an interprocedural extension of our approach along the lines of [9], and how it compares to other interprocedural algorithms. An implementation of our approach within the VFCS system [1] is in progress.

Acknowledgements: We thank the anonymous referees for their valuable comments and references to related work. Moreover, we are very grateful to Shankar Ramaswamy and Prithviraj Banerjee for making their HPF redistribution library available to us.

References

1. 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.
2. F. Coelho and C. Ancourt. Optimal compilation of HPF remappings. *Journal of Parallel and Distributed Computing*, 38(2):229–236, November 1996.
3. 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.
4. 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.
5. 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.
6. J. Knoop and E. Mehofer. Distribution assignment placement: A new aggressive approach for optimizing redistribution costs. Technical Report TR 97-6, Institute for Software Technology and Parallel Systems, University of Vienna, Austria, 1997.
7. J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. In *Proc. of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI'94)*, pages 147–158, Orlando, FL, June 1994.
8. J. Knoop, O. Rüthing, and B. Steffen. The power of assignment motion. In *Proc. of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI'95)*, pages 233–245, La Jolla, CA, June 1995.
9. 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.
10. 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.
11. D.J. Palermo, E.W. Hodges, and P. Banerjee. Interprocedural array redistribution data-flow analysis. In *Proc. of the 9th Workshop on Languages and Compilers for Parallel Computing*, San Jose, CA, August 1996.
12. 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.
13. 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.