DISSERTATION

Optimization of Data Remapping in Data-Parallel Languages

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Doktors der technischen Wissenschaften unter der Leitung von

o.Univ.-Prof. Dr. Hans P. Zima
Institut für Softwaretechnik und Parallele Systeme
Universität Wien

eingereicht an der Technischen Universität Wien
Technisch-Naturwissenschaftliche Fakultät
von

Dipl.-Ing. Eduard Mehofer
Liechtensteinstr. 22, A-1090 Wien
Matr.Nr. 83 25 566
e-mail: mehofer@par.univie.ac.at

Wien, im April 1998 Dipl.-Ing. Eduard Mehofer

KURZFASSUNG

Datenparallele Sprachen wie HPF (High Performance Fortran) geben dem Benutzer die Möglichkeit, die Aufteilung der Daten auf die lokalen Speicher der Prozessoren zu spezifizieren. Da viele wissenschaftliche Applikationen typischerweise aus mehreren Programmphasen mit unterschiedlichen Aufteilungen bestehen, sind dynamische Umverteilungen erforderlich.

HPF unterstützt Umverteilungen an Prozedurgrenzen und durch die Direktiven REDISTRIBUTE und REALIGN. Doch die Kosten von Umverteilungen können aufgrund der erforderlichen Kommunikation beträchtlich sein, und die Ausführungszeit eines Programms stark beeinflussen. Deshalb ist die Eliminierung von unnötigen Umverteilungen von großer Bedeutung. Dies umso mehr, da auch gutgeschriebene HPF-Programme unnötige Umverteilungen enthalten können.

Ziel dieser Arbeit ist es, die Umverteilungskosten zu optimieren, indem wir die Anzahl der auszuführenden Umverteilungen reduzieren. Eliminierung von redundanten und toten Umverteilungen ist nicht ausreichend. Verschieben von Umverteilungen ist erforderlich, um die Anzahl der Ausführungen weiter zu reduzieren. In unserem Ansatz erreichen wir die Reduzierung von Umverteilungen, indem wir die Eliminierung von partiell toten und partiell redundanten Code kombinieren. Das beinhaltet sowohl das Verschieben von Umverteilungen aus Schleifen wie auch deren Reduzierung außerhalb von Schleifen. Wir entwickeln einen bidirektionalen und einen unidirektionalen Ansatz zur Eliminierung von partiell toten und partiell redundanten Umverteilungen und diskutieren die Unterschiede.

Es wurde gezeigt, daß die Eliminierung von partiell totem und partiell redundantem Code ein optimales Ergebnis liefert, wenn beide separat angewandt werden. In dieser Arbeit zeigen wir, daß es im allgemeinen kein optimales Ergebnis gibt, sobald beide Algorithmen kombiniert werden. Weiters zeigen wir, daß eine direkte Adaptierung des intraprozeduralen Ansatzes für den interprozeduralen Fall nicht möglich ist.

Sowohl im intraprozeduralen Fall wie auch im interprozeduralen Fall gelangen wir zu einem einheitlichen System, welches Zuweisungen wie Umverteilungen behandelt. Es wird eine Hierarchie von Algorithmen angeboten, die sich in der Mächtigkeit und Effizienz unterscheiden und entsprechend den individuellen Bedürfnissen eines Benutzers ausgewählt werden können. Die Mächtigkeit und Flexibilität unseres Ansatzes werden mit anschaulichen Beispielen gezeigt.

Abstract

The user-controlled mapping of data across the local memories of processing nodes is one of the central features of data-parallel languages like High Performance Fortran (HPF). Since many scientific applications typically consist of different computational phases owning each a best data mapping, dynamic remappings have proven useful in maintaining good data locality and workload balance.

HPF supports remappings by procedure calls and by executing redistribute/realign-directives. But remappings can be quite expensive as communication is required to migrate the array elements to their new owning processors and can significantly degrade a program's performance. Hence, elimination of unnecessary remappings is of key importance. It is essential because even well-written HPF programs may result in unnecessary remappings.

In this thesis we optimize the overall time spent for dynamic data remappings in a program run by reducing the number of executed remappings. Elimination of redundant and dead remappings does not suffice. Code motion to eliminate remappings further is required as well. Our approach eliminates unnecessary remappings by combining both partial dead code elimination and partial redundancy elimination. In this way we succeed in sinking and hoisting remappings out of loops and eliminating them on straight-line code as well. For both partial dead code elimination and partial redundancy elimination we present a bidirectional and unidirectional approach and discuss the differences.

It has been shown that partial dead code elimination and partial redundancy elimination can be solved optimally when applied separately. Here we show that in general an optimal solution does not exist any more if both algorithms are combined. Further it turns out that a straight-forward interprocedural adaption of the intraprocedural approach fails.

For the intraprocedural as well as interprocedural approach we arrive at a uniform framework which can be applied for both ordinary and mapping assignments. The framework provides a hierarchy of algorithms of varying power and efficiency supporting user-customized solutions. The power and flexibility of our approach are demonstrated by illustrating examples.

Acknowledgments

I would like to thank my advisor, Professor Hans P. Zima, for his support and advice throughout the thesis and for providing the opportunity to work in the stimulating research environment of his Institute. Further I am grateful to my co-advisor, Professor Manfred Brockhaus, who gave me a first introduction on compilation techniques while I was an undergraduate student and who supervised my diploma thesis.

Throughout the course of my research many people have contributed to this work. I would especially like to thank Jens Knoop from the University of Passau, with whom I have had numerous discussions on analysis that have resulted in joint publications and helped to improve this thesis considerably. From our research group I would like to thank Siegfried Benkner for valuable discussions on HPF and for the support he gave me on our parallelization system VFC. Further I am indebted to Thomas Fahringer for discussing performance related issues, especially strategies for optimizing communication. Finally, I am grateful to Siegfried Benkner and Bernd Wender for partially proofreading this work and to the latter for being such a helpful and stimulating officemate.

Last, but by no means least, I am particularly indebted to my parents, Franz and Anna Mehofer, for supporting my academic career.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

High Performance Computing Systems (HPCs) represent the leading edge technologies with respect to computational power. HPCs are indispensable for many large-scale, scientific applications which simulate complex processes occurring in nature, industry, and socio-economic systems. Examples of such simulations include weather modeling and forecasting, behavior of subatomic particles, crash-tests in automobile industry, optimizing combustion engines, aerodynamic behavior of airplanes, study of chemical phenomena, or financial modeling (e.g. see [8, 14, 37, 75, 64]). Solving these problems requires appropriate hardware and software support.

1.1.1 Hardware Architectures of HPCs

From the architectural point of view HPCs offer computational concurrency and may take different forms. Above all, parallel computers may differ in control mechanism, memory organization, and interconnection network. A categorization of computers based on the control mechanism classifying them as SISD, SIMD, and MIMD has been introduced by Flynn [23]. Traditional sequential computers have a *single instruction stream, single data stream* (SISD) architecture. Computers with a single control unit but multiple processing units are referred to as *single instruction stream, multiple data stream* (SIMD) computers. Examples of SIMD parallel computers include Illiac IV, CM-2, MasPar MP-1, and MasPar MP-2. Computers with multiple processing nodes which are independent and are capable of executing different programs are called *multiple instruction stream, multiple data stream* (MIMD) computers. Examples of MIMD parallel computers include nCube, Intel iPSC, CM-5, Paragon XP/S, Meiko/Quadrics CS-2, SGI/Cray Origin 2000, SGI/Cray T3E, and Convex/HP Exemplar.

A just as important characteristic of parallel computers is the memory organization. We may distinguish between *shared-memory*, *distributed-memory*, and *virtual shared-memory* architectures. In a shared-memory architecture the memory is equally accessible by all processing nodes. In distributed-memory architectures each processing node has its own *local memory* which is only accessible to that processor. *Message passing* is used to exchange data between two processing nodes. Virtual shared-memory systems have a physically distributed-memory, but the operating system supports non-local accesses, i.e. no explicit message passing is required. Of course, accessing remote memory takes more time than accessing local memory. Depending on the memory access behavior, virtual shared-memory architectures can be categorized further. If the access time to any memory in the system is identical, they are called *uniform memory access* (UMA) machines. If the access time may vary depending on the memory location, they are called *non-uniform memory access* (NUMA) machines.

In this work we focus on the optimization of communication. The problem we describe here shows up primarily for distributed-memory MIMD systems which are often referred to as *distributed-memory multiprocessing systems* (DMMPs). But the problem is not restricted to DMMPs. In fact for NUMA machines data locality is an issue also which the programmer may want to influence and handle explicitly. In those cases, again, explicit data motion may occur in the program which can be subject to optimizations.

1.1.2 Programming of HPCs

Basically reduced execution times can be achieved on HPCs by exploiting two kinds of parallelism: *data parallelism* and *task parallelism*. Data parallelism means that portions of data are assigned to processing nodes and each node performs identical computations on its local data. Obviously, data parallelism is not appropriate for all problems (it plays an important role for scientific applications). Task parallelism refers to the execution of different instruction streams. Note that both kind of parallelisms may occur in an application and can be utilized by a program.

Parallel programming models have been developed which assist the programmer in writing applications for HPCs. Generally, we may distinguish between *explicit parallel programming* and *implicit parallel programming*. Explicit parallel programming directly reflects the machine model. Hence, this programming model is rather low level. A representative of that programming class is the *message-passing programming model*. Usually this paradigm is supported by using a standard sequential programming language, augmented with message passing constructs. The user is forced to deal with all aspects of mapping data and work to processors and controlling the program's execution. Several libraries have been developed to support the message passing paradigm: PARMACS [39], PVM [78], and the Message Passing Interface Standard (MPI) [27]. Although this approach leads to slow software development cycles and is often compared to assembly language programming for sequential machines, it is widely used since in many cases it is computationally most efficient.

Implicit parallel programming provides a high-level view of parallelism for the programmer and places a majority of the burden of parallelization on the compiler. Representatives of that class are *data parallel programming languages* which are based upon the *data parallel programming model*. Those languages enable the programmer to write programs in much the same way as they are accustomed to on sequential machines. The similar way of programming is achieved by offering a global address space, providing language support for mapping data onto processors, and assuming a single thread of control. Examples of data parallel languages are HPF [25], Fortran D [30], and Vienna Fortran [83]. Usually, data parallel programs are translated into a *single program, multiple data* (SPMD) code. In that model each processing node executes the same (parameterized) code in a *loosely synchronous* fashion. Synchronization and exchange of data is handled by the compiler by inserting appropriate code.

Languages providing support for managing task parallelism at a high level include LINDA [3], Strand [29], CC++ [11], and Fortran M [28]. Recently, a major research effort has been put on the issue of integrating task and data parallelism. Examples include Orca [7], Fx [76, 77], Approved Extensions for Data and Task Parallelism in HPF 2.0 [25], and Opus [12].

In this work we develop compiler optimizations for reducing remappings in data parallel languages. Although we use the de-facto standard HPF as base language, the problem is relevant for other data parallel languages, as well, if data remappings are supported. In the following section we motivate the use of dynamic remappings in data parallel applications.

1.2 Motivation of Dynamic Data Remappings in Data Parallel Languages

One of the main features of those data placement languages is the user-controlled mapping of data across the local memories of the processing nodes. Since data mapping has a crucial impact on runtime performance, a change of a mapping may be necessary in order to prevent workload imbalance or bad data locality for varying computational phases. Hence, most data parallel programming languages support dynamic data remappings. HPF allows to change the mapping of an array at procedure boundaries and by executing REDISTRIBUTE or REALIGN directives.

Consider a multidisciplinary application which is constructed by putting modules from a variety of related scientific disciplines together. Each module owns a best data mapping. Hence, remappings are required between the individual phases. Similar issues arise in multidisciplinary optimization. For example in modern aircraft design [12], one has a variety of interacting disciplines: aerodynamics, propulsion, structural analysis and design, and so forth. The various disciplines form a single application with an overall optimization loop of the design process whereby remappings might be necessary between the disciplines. The optimization cycle continues until the process converges with a final configuration.

Another field where remappings have proven useful are computational kernels like parallel linear algebra routines [69], 2D-FFT (two-dimensional Fast Fourier Transform) [34], or ADI (Alternate Direction Implicit) [60], a partial differential equation (PDE) solver. Figure 1.1 shows the structure of the ADI computational kernel. The kernel consists of forward and backward sweeps along the rows followed by forward and backward sweeps along the columns. Hence, a row-wise data distribution requires communication for the sweeps along the columns but none along the rows while a column-wise data distribution requires communication vice versa. By choosing a row-wise distribution for the sweeps along the rows and a column-wise distribution for the sweeps along the columns and redistributing the arrays between the two phases, no communication and synchronization is required within the sweeps at the expense of the remapping overhead.

(a) ADI main loop.

```
REAL, DIMENSION (N,N) :: x,a,b
...
DO iter=1,maxiter
   CALL rows(x,a,b)
   CALL columns(x,a,b)
END DO
...
```

(b) Subroutines performing the sweeps.

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

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

Figure 1.1: ADI integration kernel.

1.3 Problem Statement

Remappings can be quite expensive, since usually big amounts of data are communicated in order to migrate data elements to their new owning processors. Execution of unnecessary data remappings can significantly degrade a program's performance. Remappings should be used by a programmer carefully. But even well-written HPF programs may result in unnecessary remappings as shown in the Example 1.3.1 (remappings caused by an HPF directive or procedure call are written as comments immediately after that construct).

Example 1.3.1 Unnecessary remappings in well-written HPF.

```
      PROGRAM ONE
      REAL A(N,N),B(N,N),U(N)
!HPF$ PROCESSORS R(M)
!HPF$ DYNAMIC, DISTRIBUTE(BLOCK,*) ONTO R :: A
```

```
      !HPF$ DYNAMIC, ALIGN WITH A :: B
      !HPF$ DISTRIBUTE (BLOCK) :: U
            INTEGER I,J
            INTERFACE
              SUBROUTINE FOO(A)
              REAL A(:)
      !HPF$   DISTRIBUTE A (CYCLIC) ONTO *
              END SUBROUTINE
            END INTERFACE
            ...
(1)   !HPF$ REDISTRIBUTE A(*,BLOCK) ONTO R
            !  array A and B is remapped
(2)          DO I=1,N
              DO J =1,N
                 :  array A used only
              END DO
            END DO
            ...
(3)   !HPF$ REALIGN B(i,j) WITH A(j,i)
            !  array B is remapped
(4)   !HPF$ REDISTRIBUTE A(BLOCK,*) ONTO R
            !  array A and B are remapped
(5)         DO I=1,N
              DO J = 1,N
                 :  arrays A and B used
              END DO
            END DO
            ...
(6)         DO I=1,10
              !  remap U to cyclic
              CALL FOO(U)
              !  remap U to block
            END DO
```

In Example 1.3.1 array B is aligned with array A reflecting the access patterns of most computational kernels of program ONE. The REDISTRIBUTE directive at line (1) results in a remapping of arrays A and B. But the remapping of array B is dead, since B is not accessed in the loop and realigned at line (3). With regard to loop (5), a realignment and redistribution at lines (3) and (4), respectively, are performed. The realignment results in a remapping of array B which is dead, since the REALIGN directive is immediately followed by a REDISTRIBUTE directive which establishes the proper mapping for the loop. Note that an interchange of the REDISTRIBUTE and REALIGN directive does not change the situation and also results in an unnecessary remapping of array B. Finally, consider the subroutine call FOO within loop (6). Since array U is statically distributed by BLOCK and the dummy argument of subroutine FOO requires a CYCLIC distribution, array U has to be remapped just before and just after the subroutine call in order to establish the requested mapping and to restore the original one, respectively. This results in two remappings within each loop iteration. But a remapping of array U to CYCLIC just before the loop and restoring the original mapping to BLOCK just after the loop would be sufficient.

Hence, it is essential to remove unnecessary remappings. In Figure 1.2 it is shown that elimination of redundant and dead remappings is not sufficient. The HPF source of Figure 1.2(a) results in remappings shown in Figure 1.2(b). Since none of the remappings of Figure 1.2(b) are dead or redundant, respectively, redundancy and dead code elimination fail and do not result in any performance win. But the remapping at the exit of subroutine F is partially dead, since it is live with respect to the else-branch of the succeeding if-statement and dead otherwise. Obviously, the number of executed remappings can be reduced by eliminating remappings on paths where they are dead. This can be achieved by sinking remappings in the direction of the control flow and placing them as specific as possible. Hence, we are moving the remapping from the exit of subroutine F into the else-branch of the succeeding if-statement as shown in Figure 1.3(b).

Now, the remapping at the entry of subroutine F is partially redundant, since it is not redundant with respect to the then-branch of the preceding if-statement and redundant otherwise. Similarly, the number of executed remappings can be reduced by eliminating remappings on paths where they are redundant. This can be achieved by hoisting remappings in the opposite direction of the control flow and placing them as general as possible. Hence, we are moving the remapping from the entry of subroutine F to the then-branch of the preceding if-statement as shown in Figure 1.3(b).

Note that the optimized program version is free of any partially redundant or dead remapping. The optimized program is optimal in the sense that the number of remappings cannot be reduced further on any program path. The optimizations performed in Example 1.3 are more powerful than loop optimizations. Loop optimizations would place remappings just after or just before the loop. But the program can be improved further by sinking the remapping into the else-branch of the succeeding if-statement or hoisting the remapping into the then-branch of the preceding if-statement.

Clearly,the presence of procedure calls shall not prevent optimizing remappings. Interprocedural optimization has to be applied as well in order to obtain “best” results.

1.4 Outline of the Thesis

In this thesis we develop optimizations which aim at reducing the number of remappings executed on some program path. Our starting point is an HPF program with mapping directives specifying the assignment of data to processing nodes. Note that the mapping directives can either be provided by the programmer or generated by automatic data distribution tools.

As we have seen in the previous section, elimination of redundant and dead remappings does not suffice. Code motion is required to reduce the number of executed remappings further. As illustrated in Figures 1.2 and 1.3, loop-invariant code motion does not suffice either. In order to exploit all possibilities for reducing the number of executed remappings we have to solve more general problems, namely elimination of partially redundant and partially dead remappings,

```
      PROGRAM TWO
      REAL A(N)
!HPF$ PROCESSORS P(M)
!HPF$ DYNAMIC :: A
      INTERFACE
        SUBROUTINE F(A)
        REAL A(:)
!HPF$   DISTRIBUTE A (CYCLIC)
        END SUBROUTINE
      END INTERFACE
      ...
      IF (···) THEN
!HPF$   REDISTRIBUTE A::(BLOCK)
        ... A(I) ...
      ELSE
!HPF$   REDISTRIBUTE A::(CYCLIC)
        ... A(I) ...
      END IF


      DO I=1,10
        CALL F(A)
      END DO


      IF (···) THEN
!HPF$   REDISTRIBUTE A::(CYCLIC(4))
        ... A(I) ...
      ELSE
        ... A(I) ...
      END IF
```

(a) HPF source program

```
      PROGRAM TWO
      REAL A(N)
!HPF$ PROCESSORS P(M)
!HPF$ DYNAMIC :: A
      INTERFACE
        SUBROUTINE F(A)
        REAL A(:)
!HPF$   DISTRIBUTE A (CYCLIC)
        END SUBROUTINE
      END INTERFACE
      ...
      IF (···) THEN
        remap2block(A)
        ... A(I) ...
      ELSE
        remap2cyclic(A)
        ... A(I) ...
      END IF

      DO I=1,10
        remap2cyclic(A)
        CALL F(A)
        remap2restore(A)
      END DO

      IF (···) THEN
        remap2cyclic4(A)
        ... A(I) ...
      ELSE
        ... A(I) ...
      END IF
```

(b) Remappings unoptimized

Figure 1.2: Code motion required: Making remappings explicit.

```
      PROGRAM TWO
      REAL A(N)
!HPF$ PROCESSORS P(M)
!HPF$ DYNAMIC :: A
      INTERFACE
        SUBROUTINE F(A)
        REAL A(:)
!HPF$   DISTRIBUTE A (CYCLIC)
        END SUBROUTINE
      END INTERFACE
      ...
      IF (···) THEN
        remap2block(A)
        ... A(I) ...
      ELSE
        remap2cyclic(A)
        ... A(I) ...
      END IF

      DO I=1,10
        remap2cyclic(A)
        CALL F(A)
        remap2restore(A)
      END DO

      IF (···) THEN
        remap2cyclic4(A)
        ... A(I) ...
      ELSE
        ... A(I) ...
      END IF
```

(a) Remappings unoptimized

```
      PROGRAM TWO
      REAL A(N)
!HPF$ PROCESSORS P(M)
!HPF$ DYNAMIC :: A
      INTERFACE
        SUBROUTINE F(A)
        REAL A(:)
!HPF$   DISTRIBUTE A (CYCLIC)
        END SUBROUTINE
      END INTERFACE
      ...
      IF (···) THEN
        remap2block(A)
        ... A(I) ...
        remap2cyclic(A)
      ELSE
        remap2cyclic(A)
        ... A(I) ...
      END IF

      DO I=1,10
        CALL F(A)
      END DO

      IF (···) THEN
        remap2cyclic4(A)
        ... A(I) ...
      ELSE
        remap2restore(A)
        ... A(I) ...
      END IF
```

(b) Remappings optimized

Figure 1.3: Code motion required: Making optimization explicit.

which capture elimination of (partially) useless remappings uniformly in straight-line code as well as in loops.

Remapping analysis is difficult in HPF for two reasons. First, HPF provides a two-level mapping of data objects to processing nodes. Data objects are aligned relative to one another and then distributed onto the processing nodes. The maintainance of alignment relationships can cause severe problems and disables applying classical analyses directly to remapping problems. Second, the possibility to specify remappings in HPF in several ways (e.g. by directives like (re-)align and (re-)distribute or by procedure calls) make a uniform treatment difficult.

In order to solve both problems we introduce *mapping assignments* which denote the association of mappings with arrays. In an initial phase, the compiler generates an intermediate program representation by introducing mapping assignments whenever a remapping takes place in the HPF program. The introduction of mapping assignments makes remapping analyses amenable to classical analysis problems. Reduction of data remappings is equivalent to reducing mapping assignments.

Important for eliminating mapping assignments is the observation that remappings inside loops like in Figure 1.3 can be moved out of the loop by applying partial dead code elimination followed by partial redundancy elimination. Unlike many other optimization problems, we have to combine both showing up new difficulties.

Based on the work of Morel/Renvoise [65] and Dhamdhere et al. [19] who pioneered bidirectional partial redundancy elimination, we defined bidirectional partial dead code elimination and applied it to mapping assignments. However, the approach of being minimal invasive and to move mapping assignments only in case of a detected performance win, may hurt and we may miss opportunities for eliminating remappings.

Knoop, Rüthing, and Steffen showed [53, 54] that partial dead code elimination and partial redundancy elimination can be solved "optimally" when applied separately. But what happens, if both optimizations are combined? First we discuss these optimality issues in the intraprocedural case and show that optimality gets lost. The reason is that an "optimal" solution does not exist in general any more. After a detailed discussion of optimality we show that a straight-forward adaption of the intraprocedural approach fails interprocedurally. Interprocedurally an additional constraint is required in order to overcome anomalies and to allow a uniform treatment of ordinary and mapping assignments. We remark that combining partial dead code and partial redundancy elimination and the results achieved are not restricted to mapping assignments but may be applied in other contexts as well.

Finally, we present a powerful and flexible approach for eliminating mapping assignments (and ordinary assignments) resulting in a hierarchy of algorithms of varying power and efficiency fitting a user's individual needs.

The thesis is organized as follows.

Chapter 2 presents the language model and gives a survey of the HPF mapping directives. Modeling the association of mappings by mapping assignments is motivated and the semantics of mapping assignments as well as remappings is discussed.

Chapter 3 presents the compilation model and describes the realization of mapping assignments. Insertion of mapping assignments is discussed for each statement and directive which may cause an assignment of a mapping to an array.

Chapter 4 discusses intraprocedural elimination of mapping assignments. First a bidirectional approach is developed with the goal to be minimal invasive. Since beneficial optimizations may be missed, we drop the minimal invasive conditions and try to obtain "optimal" solutions. Optimality issues are discussed in detail. Finally we arrive at a powerful and flexible framework for eliminating mapping assignments.

In Chapter 5 the intraprocedural approach is adapted interprocedurally. It is shown that a straight-forward adaption may result in severe anomalies which can be avoided by defining a constraint which allows also a uniform handling of both mapping and ordinary assignments. Similarly to the intraprocedural case, we arrive again at a flexible framework.

Chapter 6 presents practical experiences and discusses the effectiveness of our approach, respectively.

Finally, we discuss the related work in Chapter 7 and draw our conclusions in Chapter 8.

Chapter 2

Assignment of Mappings to Arrays

In this chapter we discuss the association of data mappings with arrays in HPF and motivate the benefit of modeling such associations by mapping assignments. First we introduce the language model underlying this work.

2.1 Language Model

The fundamental model of parallelism in HPF is that of single-threaded data-parallel execution with a globally shared address space. Fortran array statements and loops are natural ways of specifying data parallel computation. Since data locality is critical in achieving good performance on a high-performance computer, one of the key features of HPF is the facility for user-defined data mappings. In this section we describe the concepts underlying the mapping of data objects. Since in our discussion data objects are typically arrays, we often use the term "array" synonymously for data object. The concepts can be applied straight-forward for scalars.

2.1.1 Basic Concepts

Definition 2.1.1 Index Domain

An **index domain** *of rank n is any set* $\mathbf{I}$ *that can be represented in the form* $\mathbf{I} = \mathbf{X}_{i=1}^{n} D_i$*, where* $\mathbf{X}$ *denotes the Cartesian product with $n \geq 1$ and for all $i, 1 \leq i \leq n$, D_i is a non-empty set of integers. D_i is of the form $D_i = [l_i : u_i]$ with $l_i \leq u_i$ and $[l_i : u_i]$ denoting the sequence of numbers $(l_i, l_i + 1, \ldots, u_i)$. l_i and u_i are called the lower and upper bound of dimension i, respectively.* ■

Definition 2.1.2 Array Index Domain

Let A be an arbitrary array.

1. *A is associated with an index domain* $\mathbf{I}^A$. [1]

[1] Whenever A is implied by the context, the superscript may be omitted. Analogous conventions hold for all similar cases.

2. *$\mathcal{E}^A$ is the* **set of elements** *of A. The elements are scalar objects.*
3. *$index^A : \mathcal{E}^A \rightarrow \mathbf{I}^A$ is a function establishing a one-to-one correspondence between $\mathcal{E}^A$ and $\mathbf{I}^A$. For every array element $e \in \mathcal{E}^A$, $index^A(e)$ is called the* **index** *of e.*

■

Definition 2.1.3 Index Mapping

An index mapping ι from an index domain $\mathbf{I}$ to an index domain $\mathbf{J}$ is a total function that maps each element of $\mathbf{I}$ to a non-empty set of elements of $\mathbf{J}$.

$$\iota : \mathbf{I} \rightarrow \mathfrak{P}(\mathbf{J}) - \{\phi\},$$

where $\mathfrak{P}$ denotes the power set. ■

2.1.2 Processors

The set of physical processors $\mathbf{P}$ on which a program is executed is represented by rectilinear processor arrangements which provide a means of naming and accessing individual processors and subsets thereof. We will use the notational conventions introduced for arrays above, in particular, for a processor arrangement R, $\mathbf{I}^R$ denotes the associated index domain, and $index^R : \mathbf{P} \rightarrow \mathbf{I}^R$ the function mapping physical processors to their indices.

2.1.3 Data Mapping

A mapping of an array assigns each array element to one or more processors, which become the owners of the element, and, in this capacity, store the element in their local memory. In HPF the model for assigning data objects to processor memories is a two-level mapping: first data objects are *aligned* relative to one another and, then, *distributed* onto the processors. Both alignment and distribution are modeled as index mappings between index domains.

Definition 2.1.4 Alignment

Let A denote an arbitrary array and U the align-target. An alignment of an array A with respect to U is modeled as index mapping from the index domain of array A to the index domain of U, and is denoted by

$$\alpha_U^A : \mathbf{I}^A \rightarrow \mathfrak{P}(\mathbf{I}^U) - \{\phi\}.$$

■

Array A is called the *alignee*. In HPF an align-target may be a data object or a template. Whenever an array is specified as align-target, the template of the ultimately aligned array is used. By default, an object is ultimately aligned with itself.

Definition 2.1.5 Distribution

Let U denote a distributee and R a processor arrangement. A distribution of U with *respect to a processor arrangement R is modeled as an index mapping from the index domain of U to the processor index domain, and is denoted by*

$$\delta_R^U : \mathbf{I}^U \rightarrow \mathfrak{P}(\mathbf{I}^R) - \{\phi\}.$$

■

In HPF the distributee may be a data object or a template. Whenever an array is specified as distributee, the template of the ultimately aligned array is used. Because of the close relationship between arrays and templates, we often speak loosely of the distribution of arrays.

2.1.3.1 Mapping of Arrays

The application of alignment α_U^A to an environment that specifies a data array A and an align-target U followed by the application of a distribution δ_R^U to distributee U and processor arrangement R yields a **mapping** of array A to processor arrangement R, denoted by

$$\mu_R^A : \mathbf{I}^A \rightarrow \mathfrak{P}(\mathbf{I}^R) - \{\phi\}.$$

A mapping of an array A with respect to a processor arrangement R can be described by means of the tuple $(\mathbf{I}^A, \mathbf{I}^R, \mu_R^A)$.

Definition 2.1.6 Construction of a Mapping
If α_U^A and δ_R^U are given as above, then μ_R^A can be constructed as follows. For each $\mathbf{i} \in \mathbf{I}^A$:

$$\mu_R^A(\mathbf{i}) := \bigcup_{\mathbf{j} \in \alpha_U^A(\mathbf{i})} \delta_R^U(\mathbf{j}).$$

■

2.1.3.2 Statically and Dynamically Mapped Arrays

In HPF there is a clear separation between arrays that have a fixed mapping and arrays whose mapping may change. In case of *statically mapped arrays*, the association of a mapping with an array remains invariant for the whole lifetime of the array. The mapping of *dynamically mapped arrays* may change at runtime (i.e. the array is *remapped*) by executing redistribute/realign-directives or on the return from procedure calls if the mapping of the dummy argument is inherited. Note that remappings are not limited to dynamically mapped arrays, but may occur at subprogram boundaries also if the mapping of the actual argument does not match the mapping of the dummy argument, independently whether the arrays are statically or dynamically mapped.

2.2 Specification of Data Mappings in HPF

HPF provides a two-level mapping of data objects to abstract processors. First the data objects are *aligned* relative to one another and then *distributed* onto a rectilinear arrangement of abstract processors which provide a logical view of the parallel machine. The mapping of abstract processors to physical processors depends on the HPF implementation. The combination of alignment and distribution determines the relationship between a data object A and a processor arrangement R and is called the *mapping* of A with respect to R and is denoted by μ_R^A (cf. Section 2.1.3).

In the Approved Extensions of HPF the mapping model has been extended by *dynamically mapped* arrays. Whereas for *statically mapped* arrays the association of a mapping with an array remains invariant for the whole lifetime of the array once it has been established, the mapping of a *dynamically mapped* array may be changed. In HPF dynamically mapped arrays are declared with the keyword DYNAMIC. Executable directives (REALIGN and REDISTRIBUTE) are provided for changing the mappings dynamically. In the following we will consider statically as well as dynamically mapped arrays.

2.2.1 Alignment

Alignment groups data objects relative to one another. The idea is that operations between aligned data elements which reside on the same processor are likely to be executed more efficiently. Although in most cases the same effect can be achieved with distribution as well, alignment offers frequently for the user a more convenient way and tells the compiler explicitly which elements will reside on the same processor. HPF provides ALIGN/REALIGN directives to connect an *alignee* with an *align-target*. We model an alignment as an index mapping α between two index domains (cf. Section 2.1.3).

Alignment relationships have to be maintained. If an align-target is redistributed, all aligned arrays have to be remapped as well such that the aligned data elements reside on the same processor again. If an array A is aligned with an array B which is aligned to C, array B only serves as an intermediary object. Array A is said to be *immediately aligned* with B but *ultimately aligned* with C. Hence, alignment relationships form a tree with every object being ultimately aligned with the root. Since only ultimate alignment is relevant, every object is related directly to the root forming a tree of height 1. A redistribution of the root object results in a remapping of the leaves. The following example shows some alignments.

Example 2.2.1 ALIGN and REALIGN directive.

```
            REAL A(N),B(N),X(N),Y(N)
      !HPF$ DYNAMIC :: A,B,X,Y
      !HPF$ DISTRIBUTE (BLOCK) :: X
      !HPF$ DISTRIBUTE (CYCLIC) :: Y
(1)   !HPF$ ALIGN WITH X :: B
(2)   !HPF$ ALIGN WITH B :: A
            ...
(3)   !HPF$ REALIGN WITH Y :: B
```

In the above example arrays X and Y are root objects of alignment trees and A and B are aligned arrays. Since array A is aligned with B (line (2)) which is in turn aligned with X (line (1)), A and B are ultimately aligned with X. The realign directive (line (3)) has the effect that array B is detached from X and ultimately aligned with Y which results in a remapping of array B. Note that the realignment of B has no effect on array A.

2.2.2 Distribution

The distribution partitions the data objects among the processors. We model a distribution as an index mapping δ (cf. Section 2.1.3). HPF provides the DISTRIBUTE/REDISTRIBUTE directives together with several mapping strategies for assigning the data elements to the processors. Regular distributions are supported by BLOCK(n) and CYCLIC(n). Both can be defined easily with closed forms. Support for irregular distributions is provided with GEN_BLOCK and INDIRECT which require both an integer mapping array as argument. For GEN_BLOCK the integer array contains at the *i-th* position the size of the segment which shall be stored on the *i-th* processor of the processor arrangement. In case of INDIRECT the integer array specifies the target processor of each individual element of the array dimension being distributed.

From the compiler point of view there is an important difference between regular and irregular distributions. Regular distributions (BLOCK(n), CYCLIC(n)) can be represented very easily making efficient comparisons at runtime possible, e.g. in order to skip redundant remappings or to test whether the mapping of an actual argument matches the mapping of the corresponding dummy argument. Accessing data elements of an irregular distributed array (GEN_BLOCK, INDIRECT) generally requires to access the mapping array. Since a change of the mapping array must not affect the distribution, a copy of the mapping array has to be maintained by the runtime system. This implies two important drawbacks. Firstly, duplicating the mapping array may result in considerable memory consumption. Secondly, the comparison of irregular distributions can be very expensive, since all values of the mapping arrays must be compared. Hence, compile-time analysis which detects equally mapped irregular arrays can be very effective. We will discuss this issue later in more detail.

The REDISTRIBUTE directive can be executed at any time and initiates not only a remapping of the distributee but also of all objects currently ultimately aligned with it in order to preserve the alignment relationships. Note that the use of REDISTRIBUTE directives may have the effect that at array references the mapping is control dependent as shown in the following example.

Example 2.2.2 Different mappings possible for an array reference.

```
            REAL A(N)
            INTEGER int_arr1(M),int_arr2(N)
      !HPF$ PROCESSORS R(M)
      !HPF$ DYNAMIC, DISTRIBUTE(CYCLIC) ONTO R :: A
      !HPF$ DISTRIBUTE (BLOCK) ONTO R :: int_arr2
            ...
            int_arr1 = ...
            int_arr2 = ...
            ...
            IF (···) THEN
(1)   !HPF$    REDISTRIBUTE (BLOCK) ONTO R :: A
            ELSE
```

```
(2)  !HPF$    REDISTRIBUTE (GEN_BLOCK(int_arr1)) ONTO R:: A
           END IF
(3)        ...A(i)...
(4)  !HPF$ REDISTRIBUTE (INDIRECT(int_arr2)) ONTO R :: A
(5)        ...A(i)...
```

Depending on the condition of the if-statement, array A is either redistributed by BLOCK (line (1)) or by GEN_BLOCK (line (2)). Hence, two mappings are possible for the array reference at line (3). Generally, data flow analysis is required to determine the set of possible mappings precisely. The indirect redistribution at line (4) results in a unique mapping at line (5). The analysis issues are discussed later in more detail.

2.2.3 Procedure Interface

Generally, the application of mapping directives to dummy arguments has in HPF the same semantics as for non-dummy arguments. The main difference is that dummy arguments can be additionally related to the associated actual arguments in several ways. Both DISTRIBUTE and ALIGN can be used *prescriptively*, *descriptively*, and *transcriptively*. A prescriptive directive expresses that the actual argument may be mapped differently and an automatic remapping may be necessary. A descriptive directive is a weak assertion that the actual argument is equally mapped. However, if the assertion is not true, an automatic remapping must be performed. A transcriptive directive leaves the mapping of the dummy argument unspecified and accepts the annotations of the actual argument. Additionally, the INHERIT directive tells the compiler explicitly to leave the data where they are asserting that no remapping is required (INHERIT specifies a transcriptive alignment with a distribution of DISTRIBUTE $*$ ONTO $*$).

In this way the programmer can either specify that the data shall be left in place or that during the execution of the call the data shall be automatically remapped. HPF ensures that enough information is available at compile-time such that the remapping can be done either by the caller or by the called subprogram. Statically mapped arrays accessible to the caller must have after the call exactly the same mapping as they had before the call. Only remappings of dynamically mapped arrays may be visible to the caller.

An important F90 attribute for dummy arguments is the intended use which can be utilized by HPF compilers for several optimizations (cf. Example 2.3.2 where depending on the intended use the physical data transfer can be omitted). INTENT(IN) specifies that the dummy argument shall neither be defined nor become undefined during the execution of the procedure. INTENT(OUT) specifies that the dummy argument shall be defined before a reference to the dummy argument is made within the procedure. INTENT(INOUT) specifies that the dummy argument receives data from and returns data to the invoking procedure. But note that the argument passing mechanism is not part of the language definition. A procedure call establishes an association between an actual argument and its corresponding dummy argument by which the actual argument may be accessed within the procedure. Whether this association is realized by copy-in/copy-out or call-by-reference argument passing is up to the compiler as long as the semantics is observed.

2.3 Modeling the Association of Mappings by Mapping Assignments

As shown above, the mapping of an array can be defined in several ways: (1) with specification directives (ALIGN, DISTRIBUTE), (2) with executable directives (REALIGN, REDISTRIBUTE), or (3) as a result of procedure calls. The different syntactical possibilities to specify mappings complicate analyses and optimizations. Especially the maintainance of alignment relationships causes severe problems. Since our main goal is to reduce the number of executed remappings, we have first to find out which data objects are affected by mapping directives. For distributees this can be achieved very easily: the name of the data object appears explicitly in the DISTRIBUTE or REDISTRIBUTE directive. However, the set of aligned arrays which have to be remapped as well depends on the control flow. Consider the following example.

Example 2.3.1 Redistribute directive and set of aligned arrays.

```
             REAL A(N),B(N),C(N),X(N),Y(N)
       !HPF$ PROCESSORS R(M)
       !HPF$ DYNAMIC, DISTRIBUTE(BLOCK) ONTO R :: A,B,C
       !HPF$ ALIGN WITH C :: X,Y
             ...
             IF (···) THEN
(1)    !HPF$    REALIGN WITH A :: X
(2)    !HPF$    REALIGN WITH B :: Y
             ELSE
(3)    !HPF$    REALIGN WITH B :: X
(4)    !HPF$    REALIGN WITH A :: Y
             END IF
             ...
(5)    !HPF$ REDISTRIBUTE (CYCLIC) ONTO R :: A
             ...
(6)    !HPF$ REDISTRIBUTE (CYCLIC(2)) ONTO R :: B
```

Arrays A,B,C are initially distributed by BLOCK and arrays X,Y are initially aligned with C. Depending on the if-statement, array X is realigned with A (line (1)) and array Y is realigned with B (line (2)), or X and Y are realigned the other way round (lines (3) and (4)). Note that since the arrays A, B, and C have the same distribution, the mappings of arrays X and Y are not changed by the realignments. Obviously, the redistribute directive at line (5) changes the distribution of array A from BLOCK to CYCLIC. Depending on the control flow array X or array Y has to be remapped as well in order to maintain the alignment relationship. The same holds for the second redistribute directive at line (6). Hence, either array X is remapped at line (5) and Y at line (6) or array X is remapped at line (6) and Y at line (5).

The two-level mapping with the possibility to modify dynamically alignments as well as distributions has two important impacts. On the one hand, well-known classical analyses like reaching analysis, dead code analysis, or redundant code analysis are not directly applicable to problems like reaching distributions or elimination of dead and redundant remappings. On the other hand, the different directives make a uniform treatment difficult. In order to cope with both problems, we introduce *mapping assignments* which denote the association of mappings with arrays and which are part of an intermediate representation of the HPF source program.

Definition 2.3.1 (Mapping assignments)
A mapping assignment, which is denoted by $\mathsf{map}(\mathsf{A}) := (\mathbf{I}^{\mathsf{A}}, \mathbf{I}^{\mathsf{R}}, \mu_{\mathsf{R}}^{\mathsf{A}})$*, associates mapping* $(\mathbf{I}^{\mathsf{A}}, \mathbf{I}^{\mathsf{R}}, \mu_{\mathsf{R}}^{\mathsf{A}})$ *with array* A*.*

For the ease of presentation we will often use $\mu_{\mathsf{R}}^{\mathsf{A}}$ only instead of the triplet $(\mathbf{I}^{\mathsf{A}}, \mathbf{I}^{\mathsf{R}}, \mu_{\mathsf{R}}^{\mathsf{A}})$. To improve readability further we will also use in our examples the distribution formats BLOCK, CYCLIC, GEN_BLOCK, and INDIRECT or pseudo code instead of $\mu_{\mathsf{R}}^{\mathsf{A}}$ in order to relate mappings assignments clearly to the HPF source. If the mapping of an array is specified by an alignment α, we represent the alignment directive by a mapping assignment of the form $\mathsf{map}(\mathsf{A}) := \alpha$.

Mapping related analyses and optimizations are done in two phases. In the first phase mapping assignments are inserted whenever a mapping is associated with an array. In this way all mapping changes are made explicit. In the second phase analyses and optimizations are applied to mapping assignments. The introduction of mapping assignments allows the reuse of results of classical analysis problems and makes an adaption for mapping related problems feasible. Moreover, mapping assignments are the basis for a uniform analysis framework which takes also ordinary assignments into account (as we will see later ordinary assignments have to be handled as well). Insertion of mapping assignments is described in detail in Chapter 3.

2.3.1 Association of Mappings with Arrays

Generally, the association of a mapping with an array requires some organizational overhead and a communication overhead. The organizational overhead consists basically of the local memory management for storing the owned data elements and of maintaining the runtime representation. Since data elements located on one processor under the current mapping may be located on a different processor under the newly assigned mapping, a data exchange may be required. The communication phase actually performs the physical transfer of the data elements and initializes the local segments of the processing nodes.

The association of a mapping with an array does not necessarily result in a physical transfer of data elements. Obviously, no communication is required when the specified mapping directive results in a mapping which is identical to the current one. Further, the communication can be omitted if the values of an array are undefined or dead. The following list summarizes when a physical data transfer may be omitted.

Initial Mapping. An initial mapping is associated with an array. Hence, the values of the array are undefined making a data exchange unnecessary. Initial mappings can be easily recognized at runtime.

Redundant Association. The association of a mapping is redundant, i.e. the new mapping is identical to the current one. As already mentioned, an efficient comparison of two regular mappings can be done at runtime easily (provided that the runtime system has been designed properly), whereas the effort required for INDIRECT distributions can be considerably, since it must be checked whether the mapping array has been modified or not. Redundant remapping analysis can help to reduce the effort.

Dead Array Elements. If all array elements are dead at the program point where a mapping is associated with an array, a physical data transfer is unnecessary. Analysis for identifying dead arrays has been described in [36]. Note that for dummy arguments the programmer can specify the intent-out attribute which means that the array elements can be considered dead on procedure entry.

Intent-in Attribute. If the intent-in attribute has been specified for a dummy argument, the intended use of the dummy argument is that it shall not be defined during the execution of the procedure. Hence, no physical data transfer is necessary on procedure exit if copy-in/copy-out argument passing is used.

Intent-out Attribute. If the intent-out attribute has been specified for a dummy argument, the dummy argument shall be defined in the procedure before a reference to the dummy argument is made in the procedure. Hence, no physical data transfer is necessary on the entry to the procedure if the mapping changes (independent of the copy-in/copy-out or call-by-reference mechanism).

In the following example we discuss for some mapping directives whether a physical data transfer is necessary or not.

Example 2.3.2 Mapping directives with/without physical data transfer.

```
              PROGRAM MAIN
              REAL A1(256)
              REAL, ALLOCATABLE :: A2(:)
              REAL A3(128), A4(128)
 (1)    !HPF$ PROCESSORS R(4)
 (2)    !HPF$ DISTRIBUTE A1 (BLOCK) ONTO R
 (3)    !HPF$ ALIGN A2(:) WITH A1(:)
 (4)    !HPF$ DYNAMIC A3
 (5)    !HPF$ DYNAMIC, DISTRIBUTE(BLOCK) ONTO R :: A4
              ...
 (6)            ALLOCATE A2(128)
 (7)    !HPF$   REDISTRIBUTE A3(CYCLIC(2)) ONTO R
 (8)    !HPF$   REDISTRIBUTE A4(BLOCK) ONTO R
 (9)            CALL F(A1,A2,A3,A4)
              END

              SUBROUTINE F(B1,B2,B3,B4)
              REAL, INTENT(IN) :: B1(256)
              REAL, INTENT(OUT) :: B2(:)
              REAL B3(128),B4(128)
(10)    !HPF$ DISTRIBUTE B1(BLOCK(5)) ONTO R
(11)    !HPF$ ALIGN B2(:) WITH B1(:)
(12)    !HPF$ DISTRIBUTE B3(CYCLIC(2)) ONTO R
(13)    !HPF$ ALIGN B4(:) WITH B3(:)
              ...
(14)            DO I=1,128
```

```
      B4(I) = I
    END DO
  ...
  END
```

The distribute directives at line (2) and (5) assign initial mappings and do not require communication. The alignee at line (3) is an allocatable array and, thus, the directive does not take effect immediately but is delayed until the allocate-statement at line (6) creates a new object with the specified initial mapping. Since the mapping of array A3 is not defined by a type declaration statement, a default distribution is assigned to A3. Hence, the redistribute directive at line (7) results in a physical data transfer (the communication can be skipped only if data flow analysis discovers that the values of A3 are undefined or dead). The redistribute directive at line (8) is redundant. For the dummy arguments B1 and B2 of subroutine F the intent-in and the intent-out attribute have been specified. Hence, no communication is required at the subroutine at line (9) for A1 on subroutine exit if copy-in/copy-out argument passing is used and for A2 on subroutine entry. The distribute directive for the dummy argument B3 matches the mapping of the actual argument A3. The align directive at line (13) results in a communication unless data flow analysis detects that the values of B4 are dead because of the initializing do-loop at line (14).

2.3.2 Remappings

When a mapping is associated with an array, it can be an *initial mapping assignment* or a *remapping*. In case of a remapping an already existing association is changed. However note that remappings are not limited to dynamically distributed arrays, but may also happen for statically distributed arrays automatically during the execution of procedure calls. In the following we will discuss remappings in more detail.

Remappings can be performed in HPF by executing REALIGN and REDISTRIBUTE as well as automatically during the execution of a procedure call. REALIGN and REDISTRIBUTE cancel an already existing association of a mapping with an array (if any) and establish the newly requested association. The following example shows an initial mapping assignment and a remapping.

Example 2.3.3 Initial mapping assignment and remapping.

HPF source	*intermediate representation*
`      REAL A(N)`	`REAL A(N)`
`!HPF$ DYNAMIC A`	
`!HPF$ DISTRIBUTE A(BLOCK)`	`map(A):=BLOCK   ! initial mapping`
`      ...`	`...`
`!HPF$ REDISTRIBUTE A(CYCLIC)`	`map(A):=CYCLIC  ! remapping`

In the literature the term remapping is often used in the above sense, i.e. it stands for executing REALIGN or REDISTRIBUTE resulting in a physical data transfer. However, we will use in accordance with HPF 2.0 the term remapping also at procedure boundaries if the mapping of the actual argument does not fit the evaluated annotations of the dummy argument. As already mentioned, in the F90/HPF language definition it is only said that a procedure call establishes an association between an actual argument and its corresponding dummy argument leaving it to the compiler whether to use copy-in/copy-out or call-by-reference argument passing (a detailed discussion of compiling procedure boundaries can be found in Section 3.3.3). Consider the example of Figure 2.1 where the subroutine arguments are remapped by using both copy-in/copy-out and call-by-reference. The first argument is passed by call-by-reference and the

HPF source	intermediate representation

`REAL A(N),B(N)`
`...`
`! Argument passing:`
`!   A...call-by-reference`
`!   B...copy-in/copy-out`
`CALL F(A,B)`
	`REAL A(N),B(N),tmp(N)`
	`...`
	`! entry code for A (Ā...dummy)`
	`map(A) :=` $\mu^{\bar{A}}$
	`! entry code for B (B̄...dummy)`
	`map(tmp) :=` $\mu^{\bar{B}}$
	`tmp = B`
	`CALL F(A,tmp)`
	`! exit code for A`
	`map(A) :=` μ^{A}
	`! exit code for B`
	`B = tmp`
	`deallocate(tmp)`

Figure 2.1: Remappings at procedure boundaries.

second one by copy-in/copy-out. It is assumed that both arrays A and B are statically mapped (with $\bar{A}$ and $\bar{B}$ denoting the corresponding dummy arguments), i.e. both arrays must have after the call exactly the same mapping as they had before the call. For array A call-by-reference means that A is directly passed to the subroutine. Of course, mapping assignments are required at the entry and at the exit of the subroutine call in order to establish the requested mapping and to restore the original one, respectively. If the mapping of the actual argument does not match the evaluated specifications of the dummy argument and the actual argument is passed by call-by-reference, the change of the mapping can be modeled easily with mapping assignments as shown in the example above.

This is not true for array B which is passed by copy-in/copy-out. Instead of changing an already associated mapping, a new temporary object is created with the requested mapping. On the entry as well as on the exit an array assignment is used to initialize the temporary and to update the actual argument, respectively. Hence, a remapping consists on the entry of an initial mapping assignment followed by an array assignment and on the exit of an array assignment followed by a memory release. We reflect the fact that the term remapping is used for copy-in/copy-out argument passing also by denoting the change of mappings for copy-in/copy-out by mapping assignments as well. In practice the runtime costs of an initial mapping assignment followed by an array assignment should be nearly identical to the effort required for changing an already existing mapping association (assuming reasonable implementations), since the communication which will take most of the execution time is identical.

Chapter 3

Basic Compilation of Mapping Directives

In this chapter we describe how an HPF source program is transformed into an intermediate representation containing mapping assignments. The point of the transformation is that all remappings are made explicit and described uniformly with mapping assignments. Before discussing the transformation of each directive, we introduce the compilation model.

3.1 Compilation Model

Our compilation model relies on the generation of a message-passing program based on the SPMD (single program – multiple data) programming model. In that model the data space is distributed among the processors. All processors execute a parameterized copy of the same program but only on its own data. Whenever non-local data elements are accessed, communication must be performed to exchange the data. In the following we introduce some basic terms which are needed in the context of data remappings.

Definition 3.1.1 Local Index Set and Local Segment

The set of indices $\mathbf{i} \in \mathbf{I}^A$ *of an array* A *which are mapped to a particular processor* $p \in \mathbf{P}$ *by means of a mapping* μ_R^A *is called the* **local index set** *of* p *with respect to* A *and* μ, *and is denoted by* $Local^A(p)$. *It is defined by the total function* $Local^A : \mathbf{P} \to \mathfrak{P}(\mathbf{I}^A)$ *with*

$$Local^A(p) = \{\mathbf{i} \in \mathbf{I}^A \mid index^R(p) \in \mu_R^A(\mathbf{i})\}.$$

An array element $A(\mathbf{i})$ *is said to be* **owned** *by processor* p *iff* $\mathbf{i} \in Local^A(p)$. *The data object that has to be allocated in the local memory of processor* p *to store the owned data elements of* A *is called the* **local segment** *of* p *with respect to* A, *and is denoted by* $\mathbf{Segment}_\mathbf{p}^\mathbf{A}$. ∎

In order to access the array elements of a distributed array on a particular processor, the global index has to be converted to a local one. **Index conversion functions** map global indices of the original array into local indices of the local segment, and vice versa.

Definition 3.1.2 Index Conversion Functions

A **global-to-local index conversion function** *maps the global index* $\mathbf{i} \in \mathbf{I}^A$ *of the original array* A *into the local index* $\mathbf{j} \in \mathbf{I}^{Segment_p^A}$ *of a processor* p*. It is denoted by the function* $g2l^A$: $\mathbf{I}^A \rightarrow \mathbf{I}^{Segment_p^A}$.

The inverse function $g2l^{-1}$*, which maps local indices to global ones and which is called* **local-to-global index conversion function***, is denoted by* $l2g^A$: $\mathbf{I}^{Segment^A} \rightarrow \mathbf{I}^A$.

■

The **remapping communication sets** describe which data elements are exchanged between processor pairs when a remapping takes place.

Definition 3.1.3 Remapping Send Sets

The set of indices which describe the data elements which must be sent from processor p *to processor* q *caused by a remapping of an array* A *is called the* **remapping send set** *of processor* p *to* q *with respect to* A*, denoted by* $Send^A(p,q)$*. It is defined by the total function* $Send^A : \mathbf{P} \times \mathbf{P} \rightarrow \mathfrak{P}(\mathbf{I}^A)$ *with*

$$Send^A(p,q) = \{\mathbf{i} \in \mathbf{I}^A \mid \mathbf{i} \in (Local_{new}^A(q) \cap Local_{old}^A(p))\}.$$ [1]

■

Definition 3.1.4 Remapping Receive Sets

The set of indices which describe the data elements which are received by processor p *from processor* q *by a remapping of an array* A *is called the* **remapping receive set** *of processor* p *from* q *with respect to* A*, denoted by* $Recv^A(p,q)$*. It is defined by the total function* $Recv^A : \mathbf{P} \times \mathbf{P} \rightarrow \mathfrak{P}(\mathbf{I}^A)$ *with*

$$Recv^A(p,q) = \{\mathbf{i} \in \mathbf{I}^A \mid \mathbf{i} \in (Local_{new}^A(p) \cap Local_{old}^A(q))\}.$$ [1]

■

Note that $Send^A(p,q) \equiv Recv^A(q,p)$.

[1] $Local_{old}^A$ and $Local_{new}^A$ denote the local index sets before and after the change of the mapping, respectively.

3.2 Description of Mapping Assignments

The association of mappings with arrays results in an organizational overhead and a communication overhead. The organizational overhead involves allocation of the local segment and updating of the runtime system information. The communication overhead can again be split into an indexing overhead for calculating for each processing node the send and receive index sets and the actual communication for transferring the data elements to the new owning processing nodes.

The following pseudo-code shows the realization of a mapping assignment of the form $\mathsf{map}(\mathsf{A}) := \mu$ whereby *A.Segment* denotes the pointer to the current local segment of array A and *A.MapInfo* the mapping information:

(1) On each $p \in \mathbf{P}$ do:
(2)   **begin**
(3)   allocate($\mathrm{NewSegment}_\mathrm{p}^\mathrm{A}$)
(4)   remap($\mathrm{NewSegment}_\mathrm{p}^\mathrm{A}$, μ, A.Segment, A.MapInfo)
(5)   deallocate(A.Segment)
(6)   A.Segment = $\mathrm{NewSegment}_\mathrm{p}^\mathrm{A}$
(7)   A.MapInfo = μ
(8)   **end**

First a new local segment is allocated which is passed together with the old local segment to the remapping routine remap at line (4) which actually performs the physical data transfer and initializes the new data segment. Afterwards the old local segment can be deallocated and replaced by the new local segment. Finally, the runtime information is updated.

If the mapping of an array is specified by an $\mathsf{ALIGN/REALIGN}$ directive, we have to calculate first for each array element the owning processor by applying the alignment function followed by the distribution as described in Definition 2.1.6. In the following we show the pseudo-code for $\mathsf{map}(\mathsf{A}) := \alpha_\mathsf{B}^\mathsf{A}$ with *A.AlignInfo* denoting the alignment information, *aligntarget* returning the data object to which the alignee is ultimately aligned, and *alignset* denoting the set of ultimately aligned arrays:

(1) On each $p \in \mathbf{P}$ do:
(2)   **begin**
(3) $\mu_\mathrm{R}^\mathrm{A}$ = calculate($\alpha_\mathrm{B}^\mathrm{A}$,$\delta_\mathrm{R}^\mathrm{B}$)
(4)   map(A) := $\mu_\mathrm{R}^\mathrm{A}$
(5)   aligntarget(A.AlignInfo).alignset = aligntarget(A.AlignInfo).alignset $-$ $\{$A$\}$
(6)   A.AlignInfo = $\alpha_\mathsf{B}^\mathrm{A}$
(7)   aligntarget(A.AlignInfo).alignset = aligntarget(A.AlignInfo).alignset $\cup$ $\{$A$\}$
(8)   **end**

The new mapping is calculated and assigned to at lines (3) and (4). The update of the runtime information is done at lines (5)–(7). Above all, array A has to be removed from the set of arrays aligned to the old align-target and added to the set of arrays aligned to the new align-target.

Remapping Communication

Mapping assignments are usually expensive operations mainly because they may involve communication of a big amount of data. The remapping communication consists of the following steps.

On each $p \in \mathbf{P}$ do:

1. Computation of $Send^A(p, q)$ and $Recv^A(p, q)$ respectively $\forall q \in \mathbf{P}$.
2. (a) Perform communication
   i. $\forall q \in \mathbf{P}$ with $Send^A(p, q) \neq \phi$:
      $Buffer = Segment_p^A(g2l(Send^A(p, q)))$
      send *Buffer* **to** q
   ii. $\forall q \in \mathbf{P}$ with $Recv^A(p, q) \neq \phi$:
      receive *Buffer(q)* **from** q

   (b) Assign received array elements to $Segment_p^A$
   $\forall q \in \mathbf{P}$ with $Recv^A(p, q) \neq \phi$: $Segment_p^A(g2l(Recv^A(p, q))) = Buffer(q)$ [2]

First the send and receive index sets have to be calculated on each processor. Additionally to the actual communication, a *preprocessing phase* and a *postprocessing phase* may be necessary for packing non-consecutive data elements into a buffer for transmission and for unpacking the data, respectively.

An efficient realization of the remapping communication can be critical for the performance of a program. This includes the indexing as well as the communication overhead. Efficient algorithms for reducing remapping communication have been addressed by several researchers [43, 72, 80]. One effective optimization for reducing communication overhead is latency hiding by data prefetching [59]. By splitting communication into a sending component and receiving component and overlapping communication with computation, the transfer time and possibly also the time required for copying data from user space to system space and vice versa can be hidden [82].

3.3 Insertion of Mapping Assignments

3.3.1 Distribute, Align/Realign, and Allocate

The insertion of mapping assignments for DISTRIBUTE and ALIGN directives as well as for ALLOCATE statements can be realized easily without any need for analysis. An efficient handling of REALIGN, however, may require some analysis. DISTRIBUTE and ALIGN directives in the context of dummy arguments are discussed in Section 3.3.3. The following example illustrates the insertion of mapping assignments.

[2] Note that it is assumed that the send and receive index set are ordered sets.

Example 3.3.1 Insertion of mapping assignments for DISTRIBUTE, ALIGN, REALIGN, and ALLOCATE.

HPF source

```
      REAL A(128), B(128,128), C(128)
      REAL, ALLOCATABLE :: D(:)
      PARAMETER (x=/70,32,16,10/)
!HPF$ PROCESSORS R(4)
!HPF$ DISTRIBUTE A(BLOCK) ONTO R
!HPF$ ALIGN B(:,*) WITH A(:)
!HPF$ DYNAMIC :: C
!HPF$ DISTRIBUTE(GEN_BLOCK(x)) ONTO R :: C
!HPF$ DISTRIBUTE D(CYCLIC) ONTO R
      ...
      I=...
      ALLOCATE(D(I))
!HPF$ REALIGN C(:) WITH D(:)
      ...
```

intermediate representation

```
      REAL A(128), B(128,128), C(128)
      REAL, ALLOCATABLE :: D(:)
      PARAMETER (x=/70,32,16,10/)
!HPF$ PROCESSORS R(4)
      map(A) := BLOCK
      map(B) :=  α_A^B
      map(C) := GEN_BLOCK(x)
      ...
      I=...
      map(D) := CYCLIC
      map(C) :=  α_D^C
      ...
```

The mapping directives for the statically mapped arrays A and B can be translated easily into mapping assignments. The initial mapping of the dynamically mapped array C, it is treated analogously to array A. The distribute directive for the allocatable array D does not take effect immediately and, hence, no mapping assignment is inserted. However, each time array D is allocated, a mapping assignment has to be inserted. The realign-directive is treated analogously to the align-directive. If the mapping of the align target as well as the alignment expression is statically defined, the mapping of the alignee can be calculated at compile-time. But note that mapping assignments reflect the same runtime dependencies as the corresponding directives do as shown in the following example.

Example 3.3.2 REALIGN with runtime dependencies.

HPF source	*intermediate representation*
`      REAL A(N,N), B(N)`	`      REAL A(N,N), B(N)`
`      INTEGER x(N)`	`      INTEGER x(N)`
`!HPF$ PROCESSORS R(M)`	`!HPF$ PROCESSORS R(M)`
`!HPF$ DYNAMIC :: A, B`	`!HPF$ DYNAMIC :: A, B`
`      ...`	`...`
`      IF (`*cond*`)`	`IF (`*cond*`)`
`!HPF$   REDISTRIBUTE B(CYCLIC)`	`  map(B) := CYCLIC`
`      ELSE`	`ELSE`
`!HPF$   REDISTRIBUTE B(INDIRECT(x))`	`  map(B) := INDIRECT(x)`
`      END IF`	`END IF`
`      ...`	`...`
`!HPF$ REALIGN A(:,*) WITH B(:)`	`map(A) :=` $\alpha_{\mathtt{B}}^{\mathtt{A}}$
`      ...`	`...`

Above we assumed that no array is ultimately aligned with B (REDISTRIBUTE directives are discussed in Section 3.3.2). Hence, only array B is remapped. Since array B may have a CYCLIC or INDIRECT distribution depending on the control flow, the mapping of array A which is specified by $\alpha_{\mathsf{B}}^{\mathsf{A}}$ is runtime dependent and we do not know whether the first dimension of A is distributed in a CYCLIC manner or with INDIRECT. Reaching mapping definition analysis, which is defined analogously to the traditional reaching definition problem [2], delivers the set of reaching mapping assignments and can help to suppress runtime dependencies.

3.3.2 REDISTRIBUTE Directive

There is an important difference between the REDISTRIBUTE directive and other mapping directives complicating insertion of mapping assignments for REDISTRIBUTE directives considerably. Usually a mapping directive applies to exactly one data object which appears textually in the directive (or if the attributed form of a directive is used it applies to a list of data objects which appear textually in the directive). However, a REDISTRIBUTE directive (cf. Section 2.2.2) does not only result in a remapping of the distributee but also in a remapping of the set of ultimately aligned data objects. The aligned objects do not textually appear in a REDISTRIBUTE directive and, moreover, may depend on the control flow. In this section we discuss *reaching alignment analysis* in detail which can be used to reduce the number of inserted mapping assignments at REDISTRIBUTE directives. The following example shows a control flow dependence which affects the set of ultimately aligned arrays.

Example 3.3.3 REDISTRIBUTE and set of ultimately aligned arrays.

HPF source

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

intermediate representation

```
      REAL A(N,N), B(N), C(N)
!HPF$ PROCESSORS R(M,M)
!HPF$ DYNAMIC :: A, B, C
      ...
      IF (···) THEN
          map(B) :=  α_A^B
      ELSE
          map(C) :=  α_A^C
      END IF
      ...
      map(A) := (CYCLIC,CYCLIC)
      IF (is-aligned(B,A)) map(B) :=  α_A^B
      IF (is-aligned(C,A)) map(C) :=  α_A^C
      ...
```

Depending on the control flow either array B or array C is aligned with root A as specified by the REALIGN directives. Since it is undefined at compile-time which array is actually aligned with A, the mapping assignments inserted for the alignees are guarded with masks asserting that an alignment relationship actually exists. Note that a simple strategy would be to defer the decision until runtime. However, it is one of our goals to denote explicitly where a mapping is assigned to an array in order to make mapping analyses amenable to standard analysis techniques. In the following we will distinguish between three different situations which lead to different approaches for determining ultimately aligned data objects depending on whether REALIGN directives are used in a program and depending on the places where REDISTRIBUTE directives occur and where the corresponding distributees are declared.

Case 1: No REALIGN; REDISTRIBUTE and distributee in same scoping unit

If no REALIGN directive occurs in a program and the REDISTRIBUTE directive appears in the same scoping unit as the declaration of the corresponding distributee, the set of aligned arrays is control flow independent and can be determined very easily. It is sufficient to search the specification statements of the current unit for ALIGN directives which have as align-target the distributee of the REDISTRIBUTE directive under consideration. Such a simple algorithm is possible, since at procedure boundaries data mappings may be changed only, but not alignment relationships as shown in Figure 3.1. All arrays are dynamically mapped and array B is aligned with array A. At the entry of subroutine call F the actual arguments A and B need to be automatically remapped as specified by the mapping directives for the dummy arguments X and Y. The interesting point is the return from subroutine F. Since actual argument A and the corresponding dummy argument X are dynamically mapped, the distribution of dummy argument X is assigned to A resulting in A being cyclically distributed. Similarly, both the

```
      PROGRAM MAIN
      REAL A(1024),B(1024),U(1024)
!HPF$ DYNAMIC::A,B,U
!HPF$ DISTRIBUTE(BLOCK)::A
!HPF$ DISTRIBUTE(CYCLIC)::U
!HPF$ ALIGN WITH A::B
        ...
      CALL F(A,B)
        ...
      END
```

```
contains:

      SUBROUTINE F(X,Y)
      REAL X(1024),Y(1024)
!HPF$ DYNAMIC::X,Y
!HPF$ ALIGN WITH U::X
!HPF$ DISTRIBUTE (CYCLIC(4))::Y
        ...
      END
```

Figure 3.1: Invariant alignment relationships.

actual and the dummy argument B and Y are dynamically mapped. However, the distribution of dummy argument Y cannot be assigned to actual argument B, since it would violate the alignment relationship of arrays A and B. Hence, a change of the mapping of array B to CYCLIC is required on subroutine exit. Note that the obtained set of aligned objects is an exact result, i.e. it is no conservative overestimation. Hence, for each element of the set a mapping assignment without guarding mask can be inserted.

Case 2: REALIGN used; REDISTRIBUTE and declaration of distributee appear in internal subprogram

If REALIGN directives occur in a program and the REDISTRIBUTE directive and the corresponding distributee are specified in an internal subprogram, intraprocedural analysis is sufficient to determine the set of aligned arrays. The problem of finding the set of possibly aligned arrays is identical to the problem of finding all ALIGN/REALIGN directives that are reaching some REDISTRIBUTE directive. The data flow analyses for reaching alignments is summarized in Appendix A.

Note that the obtained set of aligned arrays need not to be exact, but can be a conservative overestimation. Hence, mapping assignments have to be guarded with masks as shown in Example 3.3.3. Obviously, the set obtained by reaching alignment analysis must be a subset of dynamically mapped alignees occurring in a program. Note that conservative approximations of the set of possibly aligned arrays based on flow-insensitive information will yield in practice reasonable results in many cases.

Case 3: Otherwise

In all remaining cases insertion of mapping assignments for ultimately aligned arrays is prevented for one or the other reason. Figure 3.2 shows an example where a REDISTRIBUTE directive is not in the scope of an ultimately aligned array and, hence, a mapping assignment cannot be inserted. Variable A denotes a global variable which can be accessed by both internal subprograms F and G. The call of subroutine F causes the local variable X to be aligned with A. Hence, the REDISTRIBUTE directive in subroutine G which is called from F affects the mapping of X as well. But as the scope of variable X is limited to F, we cannot insert a mapping assignment at the REDISTRIBUTE directive.

```
      PROGRAM MAIN
      REAL A(1024)
!HPF$ PROCESSORS R(16)
!HPF$ DISTRIBUTE(BLOCK) ONTO R :: A
!HPF$ DYNAMIC :: A
        ...
      CALL F
        ...
      END


contains:

      SUBROUTINE F
      REAL X(1024)
!HPF$ DYNAMIC :: X
!HPF$ ALIGN WITH A :: X
        ...
      CALL G
        ...
      END

      SUBROUTINE G
      REAL Y(1024)
!HPF$ PROCESSORS R(16)
        ...
!HPF$ REDISTRIBUTE A(CYCLIC) ONTO R
        ...
      END
```

Figure 3.2: REDISTRIBUTE with alignees out of scope.

A simple strategy to handle such cases is to defer remappings until runtime at the price that mapping assignments do not appear explicitly in the code any more. The following pseudo-code with A denoting the array being redistributed (notation introduced in Section 3.2) shows a runtime solution for remapping all aligned arrays. Note that the set of aligned arrays is maintained by the runtime system.

(1) map(A) := $\mu_{\mathrm{R}}^{\mathrm{A}}$ /* *remap distributee* */
(2) **forall** X $\in$ A.alignset: /* *remap set of alignees* */
(3) **begin**
(4) $\mu_{\mathrm{R}}^{\mathrm{X}}$ = calculate($\alpha_{\mathrm{A}}^{\mathrm{X}}$,$\delta_{\mathrm{R}}^{\mathrm{A}}$)
(5) map(X) := $\mu_{\mathrm{R}}^{\mathrm{X}}$
(6) **end**

Whenever for some reason the insertion of mapping assignments for aligned arrays is not possible, we use the above described runtime resolution which is denoted by a mapping assignment of the form $\mathsf{map}(\mathsf{A}_{\mathsf{root}}) := \mathsf{rhs}$ where the left-hand side of the mapping assignment expresses that a root object of an alignment tree is redistributed affecting all objects of the tree.

3.3.3 Procedure Boundaries

At procedure boundaries we are facing in particular two problems. On the one hand, we have to decide whether the caller or the called procedure shall perform automatic remappings. On the other hand, procedure arguments can be passed either by copy-in/copy-out or call-by-reference. In the following we will discuss both issues.

Where to place remappings?

HPF allows at procedure boundaries to specify whether the data should be left in place or whether during the execution of a call the data should be automatically remapped. Since HPF ensures that remappings are visible to the caller at compile-time, remappings can be performed either in the caller or in the callee. Generating the remapping in the caller provides greater flexibility and offers more possibilities for optimizations as shown in the example below.

Example 3.3.4 Motion of mapping assignments.

before optimization

```
DO ...
  map(A) := BLOCK
  CALL F(A)
  map(A) := CYCLIC
END DO
```

after optimization

```
map(A) := BLOCK
DO ...
  CALL F(A)
END DO
map(A) := CYCLIC
```

Subroutine F is called within a loop with two automatic remappings just before and just after the subroutine call in order to set up the requested mapping and to restore the original one, respectively (it is assumed that array A is statically mapped and a remapping at subroutine exit is required). As we will see later, for non zero-trip loops code motion may succeed in hoisting and sinking both mapping assignments out of the loop resulting in the code depicted on the right side of the above example. Now assume that subroutine F is part of a library, i.e. only the object code is available, and the remapping is performed in the called subroutine. Obviously, the remappings cannot be moved out of the loop any more resulting in a highly inefficient code since both mapping assignments have to be performed in each iteration. But on the other hand, it may be easier to generate remappings in the called procedure as shown below.

Example 3.3.5 Evaluation of arguments.

```
      PROGRAM MAIN
      ...
      CALL F(A(I:I+J, 2*I:2*I+J), func(5))
      END

      SUBROUTINE F(X,U)
      REAL X(:,:)
      INTEGER U
!HPF$ DISTRIBUTE X(CYCLIC(U),*)
      ...
      END
```

While in subroutine F the shape of array X and the value of the block size U can be simply accessed, both have to be calculated in the caller in order to establish the requested mapping and to pass it to the callee. But it can be even more intricate. Consider the following example.

Example 3.3.6 Automatic remappings and scope.

```
      PROGRAM MAIN
      ...
      CALL F(A)
      END

      SUBROUTINE F(X)
      USE M   ! contains U
      REAL X(:,:)
!HPF$ DISTRIBUTE B(CYCLIC(U),*)
      ...
      END
```

Block size U of the block-cyclic distribution is use associated. If module M is not included in the main program, CYCLIC(U) cannot be evaluated in the caller. Moreover, the declaration of a local variable U in the main program could prevent a simple inclusion of module M.

On behalf of the greater optimizational power remappings should be performed by the caller whenever possible. But in exceptional cases remappings must be done selectively in the called procedure. If not stated otherwise, we will assume in the following that generally the *caller supports the callee*, i.e. mapping assignments are placed at the call site just before and just after the procedure call.

How to pass procedure arguments?

As already mentioned, a procedure call establishes an association between an actual argument and its corresponding dummy argument by which the actual argument may be accessed within the procedure. In HPF 2.0 [25] it is specified (p. 49): *If a remapping of the corresponding actual argument is necessary, the call should proceed as if the data where copied to a temporary variable to match the mapping of the dummy argument.* This means that the calling mechanism should behave like copy-in/copy-out, but as long as it is semantics conforming other argument passing strategies can be applied as well. The main criteria for choosing a particular argument passing strategy will be efficiency and simplicity (apart from semantically correctness). But first let us consider again the code generated for copy-in/copy-out and call-by-reference argument passing (cf. Example 2.1).

Example 3.3.7 Copy-in/copy-out and call-by-reference argument passing.

copy-in/copy-out

```
! A ... actual argument
! Ā ... dummy argument
map(tmp) := μ^Ā
tmp = A
CALL F(tmp)
A = tmp
deallocate(tmp)
```

call-by-reference

```
! A ... actual argument
! Ā ... dummy argument

map(A) := μ^Ā
CALL F(A)
map(A) := μ^A
```

The copy-in/copy-out mechanism requires at the entry (1) to allocate a temporary which is initialized with the mapping of the corresponding dummy argument and (2) to initialize the temporary as shown on the left side of the above example. At the exit (1) the actual argument must be updated and (2) the temporary deallocated. In case of call-by-reference, a reference to the actual argument is passed to the procedure which requires automatic remappings on the entry and exit of the procedure in order to establish the requested mapping and to restore the original one, respectively.

Now let us compare both strategies in terms of memory and runtime costs. Obviously, the memory costs of copy-in/copy-out are higher: since the actual argument is copied to a temporary, the memory requirement for holding the array is duplicated as long as the called procedure is active. The runtime costs of both strategies depend heavily on the fact whether automatic remappings with communication take place or not. Table 3.1 shows the data transfers required realizing particular procedure interfaces based on copy-in/copy-out and call-by-reference argument passing.

In case (1) of Table 3.1, an automatic remapping of the actual argument is performed on entry and exit of the procedure call. The runtime costs of the copy-in/copy-out and call-by-reference will be nearly the same, since the required communication is identical for both strategies and will take most of the remapping time. In case (2) of Table 3.1, an automatic remapping is performed on entry, but on exit the mapping of the dummy argument is assigned

	Procedure argument	*copy-in/copy-out*	*call-by-reference*
(1)	remapping with communication on entry and exit	2 communications	2 communications
(2)	remapping with communication on entry and none on exit	2 communications	1 communication
(3)	remapping with communication on exit and none on entry	1 copy + 1 communication	1 communication
(4)	no remapping on entry as well exit	2 copies	–

Table 3.1: Techniques for subscript expressions.

to the actual argument. Hence, for call-by-reference on exit nothing has to be done whereas for copy-in/copy-out the actual argument gets assigned the mapping of the temporary followed by a communication in order to transfer the data back to the actual argument resulting in one extra communication phase. In case (3) of Table 3.1, the effort required on procedure exit is the same for both strategies, i.e. both have to perform a communication. However on procedure entry, for call-by-reference no data transfer is required whereas for copy-in/copy-out the data have to be copied from the actual argument to the temporary (no communication necessary). What has been said about the procedure entry in case (3) of Table 3.1 applies exactly to both procedure entry and exit in case (4) resulting in two copies for copy-in/copy-out and no data transfer for call-by-reference.

If the dummy argument need not to be initialized or the actual argument shall not be updated by the dummy argument, e.g. because the programmer has specified an intended use (cf. Section 2.3.1), no data transfer is required which will result in similar runtime for both strategies. Since copy-in/copy-out argument passing results in higher memory costs and in many cases in higher runtime costs as well, *call-by-reference argument passing* will be used in the following *whenever possible.*

Note that call-by-reference argument passing can be modeled easily by mapping assignments as shown by Example 3.3.7. The copy-in/copy-out strategy, however, cannot be realized with mapping assignments alone; additional code is required.

(1)	procedure entry:	inserted code	$\mathsf{map(X)} := \mu$ $\mathsf{X} = \mathsf{A}$
(2)	procedure exit:	inserted code	$\mathsf{A} = \mathsf{X}$ $\mathsf{deallocate(X)}$

The code inserted at procedure entry/exit is modeled by mapping assignments, as well. In this way, copy-in/copy-out argument passing can be described with mapping assignments conveniently.

Chapter 4

Mapping Assignment Placement

In the intermediate program representation all remappings occurring in an HPF program are represented by mapping assignments. Hence, the problem of reducing the number of executed remappings is equivalent to the problem of reducing the number of executed mapping assignments. Eliminating redundant and dead mapping assignments does not suffice. Code motion is required as well in order to place mapping assignments on less frequently executed program paths. But let us first give some basic definitions and motivate our approach with illustrating examples.

4.1 Basic Definitions and Terminology

4.1.1 Program Flow Graphs

We represent program units by *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 (statements) ι, edges $(m, n) \in E$ the branching structure of G, and $\mathbf{s}$ and $\mathbf{e}$ the unique *start node* and *end node* of G, which are assumed to have no 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. A path of length $k - 1$ is a sequence of nodes $(n_1, ..., n_k)$ such that $(x_i, x_{i+1}) \in E$ for $1 \leq i \leq k - 1$. The set of all finite paths from m to n is denoted by $\mathbf{P}[m, n]$. For the sake of clarity, we often use for an arbitray but fixed path $p \in \mathbf{P}[m, n]$ the notation $[m, n]$. Path p without starting and ending node is denoted by $]m, n[$. All nodes of G are assumed to lie on a path from $\mathbf{s}$ to $\mathbf{e}$. Moreover, all statements occurring in a program flow graph are classified as follows: (ordinary) *assignment statements*, *mapping assignments*, subprogram calls, output operations of the form $out(t)$ forcing all operands of term t to be live, and finally the the *empty statement* skip. Conditions in if-statements or loops are modeled with output operations. Program input is modeled with assignment statements. Further, all *critical edges* which start in a node with more than one successor (branching node) and lead to a node with more than one predecessor (joining node) are assumed to be split by inserting a synthetic node (cf. Figure 4.1: edge **(1,4)** is a critical edge). This is typical for code motion transformations in order to avoid that the motion pro-

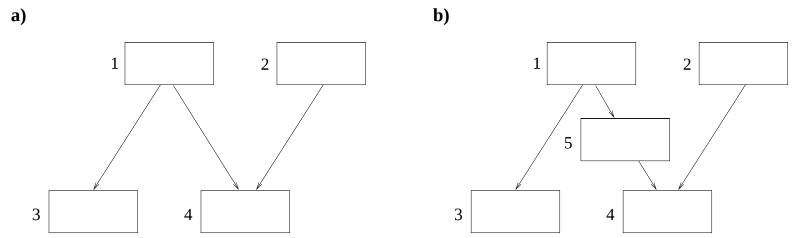


Figure 4.1: Critical edges and edge splitting.

cess is blocked. Unless stated otherwise, we restrict our attention to programs where every critical edge has been split by inserting a synthetic node. Finally, we distinguish between *assignment* and *assignment pattern*. An *assignment pattern* represents the textual form in the source program. An *assignment* is an instance of an *assignment pattern*. To clarify this point consider Figure 4.3(a). There we have two *assignments*: one at node **2** and another at node **4**. However, we have only one *assignment pattern* of the form $\mathsf{x} := \mathsf{a}+\mathsf{b}$. Let $\mathcal{P}$ denote the set of ordinary and mapping assignment patterns occurring in G, and let $\mathcal{D} \subseteq \mathcal{P}$ denote the subset of mapping assignment patterns only.

4.1.2 Data Flow Problems

Data flow problems can be classified as *forward*, *backward*, and *bidirectional* problems. A data flow problem is called a *forward* problem, if the information is propagated in the direction of control flow. A data flow problem is called a *backward* problem, if the information is propagated in the opposite direction of the control flow. If the information is propagated in both directions, then the problem is called *bidirectional*. The following data flow problems and terms are used throughout this work.

Available Assignment

An assignment pattern is called *available* at a program point p, if on every path from entry node $\mathbf{s}$ to p the assignment is performed and none of the variables occurring in the assignment are modified between the last execution of the assignment and p for each path. This means that the assignment inserted at p would deliver the same result as the last execution of the same assignment pattern on this path.

Partially Available Computation

An assignment pattern is called *partially available* at a program point p, if there is at least one path from entry node $\mathbf{s}$ to p with an execution of the assignment and none of the variables occurring in the assignment being assigned values between the last such execution on that path and p. Note that partial availability is a weaker property than availability.

Redundant Assignment

An assignment at a program point p is called *redundant*, if on every path from entry node

s to p the assignment pattern is executed and none of the variables occurring in the assignment pattern are modified between the last execution on a path and p. Note that an assignment at a program point p is redundant, if the assignment pattern is available at p.

Partially Redundant Assignment
An assignment at a program point p is called *partially redundant*, if there is at least one path from entry node **s** to p where the assignment pattern is executed and none of the variables occurring in the assignment pattern are modified between the last execution on that path and p. Note that an assignment at a program point p is partially redundant, if the assignment is partially available at p. Further note that partial redundancy is a weaker property than redundancy.

Live Variable, Dead Variable
A variable is called *live* at a program point p, if there is a use of the variable on some path from p to the exit node **e** . Otherwise, the variable is called *dead*.

Dead Assignment
An assignment $v := exp$ at a program point p is called *dead*, if v is dead.

Partially Dead Assignment
An assignment at a program point p is called *partially dead*, if there is at least one path from p to exit node **e** where the assignment is dead. Note that partially dead code is a weaker property than dead code.

Loop-invariant Computation
A computation in a loop is called *loop-invariant*, if it produces the same result on every iteration of the loop.

Useless Computation
A computation is called *useless*, if it is redundant or dead.

The terms redundant computation and available computation can be directly applied to mapping assignments making definitions for redundant and available mapping assignments not necessary. An ordinary assignment is dead depending on whether the left-hand side variable of the assignment is used or not, i.e. the assigned value is used or not. In case of mapping assignments the value assigned represents a mapping. In the following we give the definitions of *dead mapping* and *dead mapping assignment*.

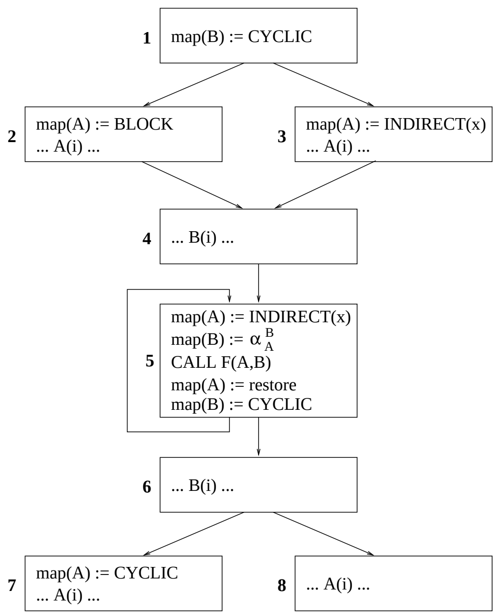
Definition 4.1.1 Dead Mapping, Dead Mapping Assignment
A mapping of a data object is called dead at a program point p if on every path from p *to the exit node* **e** *the mapping is not used. A mapping assignment is identified as dead code iff the mapping is dead.* ■

The solution of data flow equations at a flow node n is the meet over the effects under consideration of all paths going through n and is, hence, called the *meet-over-all-paths* (MOP) solution. But generally, the MOP solution cannot be computed directly, since there are in general infinitely many program paths reaching a flow node n. Instead the general iterative algorithm [38] resulting in the *maximum fixed point* (MFP) solution can be used. Kildall [47] developed a lattice-theoretic approach which was later clarified and generalized by Kam and Ullman [41, 42]. The Coincidence Theorem of Kam and Ullman [42], which gives a sufficient condition for the coincidence of the MOP and MFP solution, shows that for data flow problems in which all flow functions are distributive, the MOP and the MFP solutions are identical. All data flow problems described here can be solved easily using the general iterative approach.

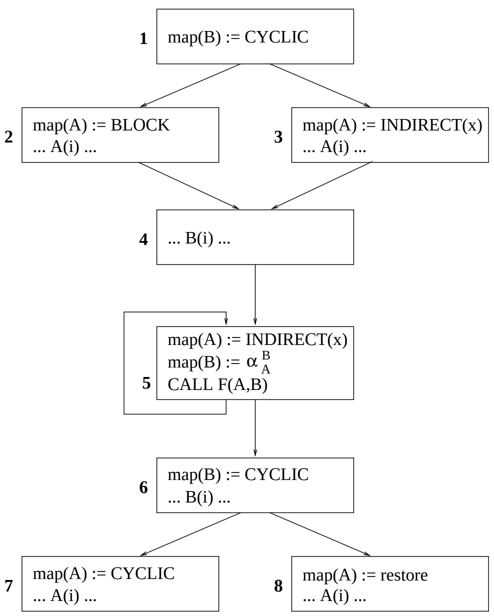
4.2 Motivating Example

As already mentioned, reducing the number of remappings at runtime is of key importance to gain efficiency. Eliminating (totally) *redundant* and *dead* mapping assignments is in general not sufficient as illustrated by the example of Figure 4.2. The example consists of a loop which contains a subroutine call F with mapping assignments just before the call to establish the requested mappings and just after the loop to restore the original ones. The first argument of subroutine F is distributed indirectly; the second argument is aligned with the first one indicated by map(B) := *aligned-to-A*. It is assumed that subroutine F leaves the mappings of both arrays unchanged. Note that none of the mapping assignments is (totally) redundant or (totally) dead. This prevents elimination. However, the mapping assignments just after subroutine F are *partially dead*, since they are dead within the loop and alive with respect to nodes **6**, **7**, and **8**. Thus, they can be placed just after the loop. While mapping assignment map(B) := CYCLIC is now totally live and reached its final position, the second mapping assignment map(A) := restore is still partially dead due to node **7**. However, further sinking of the mapping assignment to node **8** removes this partial dead code as well resulting in the flow graph shown in Figure 4.2(b). Note that now all mapping assignments are totally live. As a side effect, the mapping assignments just before subroutine F, which were required in each loop iteration to re-establish the entry mappings, are now redundant within the loop. Such effects are usually called *second-order effects* (cf. [74]). But note that the second mapping assignment is blocked by the first one because of the alignment specification. Hence, we have first to hoist mapping assignment map(A) := INDIRECT(x) to node **4**. As another second-order effect, we are now allowed to hoist mapping assignment map(B) := *aligned-to-A* to node **4** as well. While mapping assignment map(B) := *aligned-to-A* reached already its final position, the mapping assignment map(A) := INDIRECT(x) is still partially redundant due to node **3**. Hoisting further of the mapping assignment to node **2** removes the partial redundancy. The final result is shown in Figure 4.2(c). Note that none of the mapping assignment are partially dead or partially redundant any more. The example leads us to some important observations.

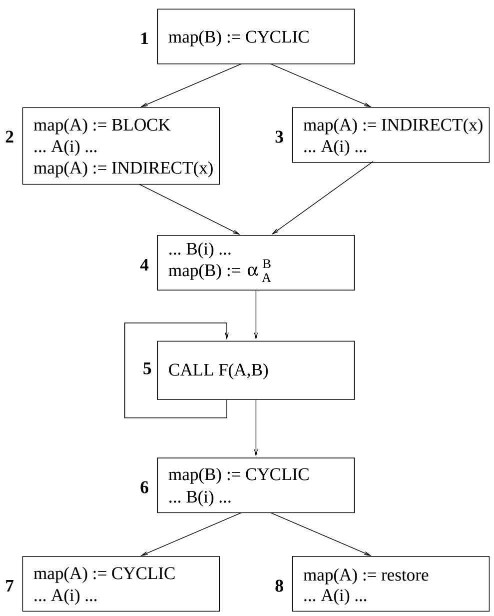
Observation 1. Morel and Renvoise [65] proposed to do elimination of redundant code and loop-invariant code motion at the same time by solving a more general problem, namely partial redundancy elimination. As shown in our first motivating example, loop-invariant computations can be seen as special cases of partially redundant computations. However, partial redundancy elimination does not suffice in our case. To cope with procedure calls inside loops which require remappings just before the call to establish the requested



a) Original flow graph



b) After partial dead code elimination



c) Optimized flow graph

Figure 4.2: First motivating example.

mappings and just after the call to restore the original one as shown in node **5**, we have to sink mapping assignments and place them after the loop. Partial dead code elimination exactly performs this code motion. Hence, partial dead code elimination followed by partial redundancy elimination succeeds in moving all mapping assignments out of the loop as shown in Figure 4.2(c).

Observation 2. The mapping assignments are not placed just before the loop and just after the loop as loop optimizations would do resulting in still partially redundant and partially dead mapping assignments. Since a more general problem is solved, the optimization is more powerful as well arriving at a program which is free of any partially redundant or dead mapping assignment.

Observation 3. Partially dead mapping assignments have to be executed in any case. In practice, it is not possible to specify a guarding mask which ensures that a mapping assignment is only executed if it is alive. Such a guarding mask would require to summarize all branching conditions of all paths where the mapping is used in order to test in advance whether such a path will be taken.

Observation 4. In general, the execution of partially redundant mapping assignments can be prevented by guarding masks. Usually this will be quite efficient for regular distributions, however, the evaluation of the mask can be quite expensive in case of INDIRECT distributions, since it must be checked at runtime whether the mapping array, which has the same size as the mapped array, has been modified or not. In our example data flow analysis found out that array x is not changed making a guarding mask unnecessary.

4.3 Bidirectional Mapping Assignment Placement

As we have seen in the previous section, partial dead code elimination followed by partial redundancy elimination is necessary above all for moving mapping assignments out of loops. In this section we present bidirectional partial redundancy and partial dead code elimination of mapping assignments and the analysis framework composed of both steps (cf. [61]). Bidirectional partial redundancy elimination of mapping assignments has been adapted from [19]. We developed bidirectional partial dead code elimination dual to bidirectional partial redundancy elimination.

4.3.1 Bidirectional Partial Redundancy Elimination

The idea of partial redundancy elimination [65, 16, 18, 19] is to move assignments in the opposite direction of the control flow and to place them in a more general context maximizing in this way the potential of redundant code which can be eliminated thereafter. This transformation idea is illustrated in Figure 4.3. The assignment x:=a+b at node **4** is partially redundant, since it is redundant with respect to the assignment at node **2** and not redundant on paths entering node **4** via edge **(1,4)**. Hoisting the assignment x:=a+b from node **4** to node **1** would eliminate the partially redundancy. However, insertion of the assignment at node **1** is

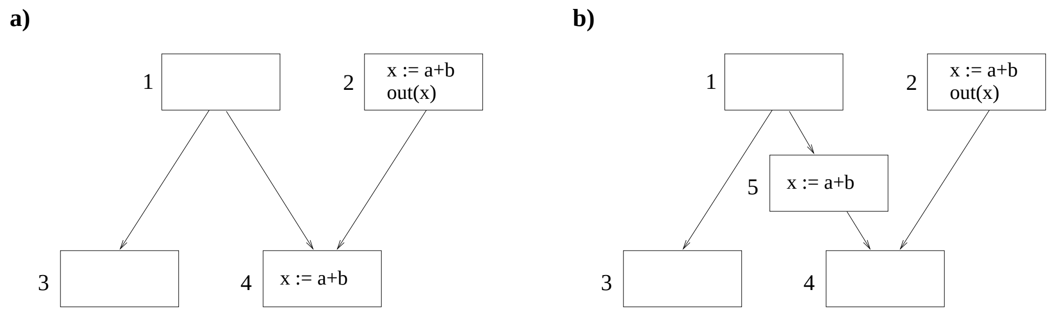


Figure 4.3: Hoisting and edge placement.

not allowed, since a new computation would be introduced on paths entering node **3** via edge **(1,3)**. But the partial redundancy can be eliminated by inserting a synthetic node on critical edge **(1,4)** as shown in Figure 4.3(b). Such selective insertion of synthetic nodes is also called *edge placement* contrary to *edge splitting* which is done for all critical edges.

Let us first discuss the *safety issues* for assignment hoisting. Assignment hoisting is called *safe*, if no assignment is inserted on a path where it was not present. More technically, nodes satisfying the safety condition for the insertion of an assignment are said to be *very busy* at that point. An assignment α is called very busy at a program point p if on each path starting from p, the assignment α will be executed before any of its operands are changed or the left-hand side of α is changed or used. Table 4.1 displays the data flow system for assignment hoisting. The data flow equations are based on the local predicates **LocHoist** and **LocBlock**. $\mathbf{LocHoist}_n(\alpha)$ specifies whether there is a *hoisting candidate* of α inside basic block n. Hoisting candidates are mapping assignments inside basic blocks which are not blocked by a preceding instruction of the same basic block (such mapping assignments are also called *upwards exposed*). Basically, a mapping assignment of the form map(A) := rhs is blocked by any occurrence of A and any definition of a variable used in the rhs-term (for example an assignment to variable K with K denoting the block size of the block-cyclic distribution CYCLIC(K)). In addition, procedure calls are considered blockades for mapping assignments (the interprocedural case is discussed later). $\mathbf{LocBlock}_n(\alpha)$ specifies whether assignment pattern α is blocked by instructions inside basic block n and, hence, cannot be hoisted through node n.

The **PPIN/PPOUT** equations assure the safety of code motion and are identical to the very busy data flow problem. $\mathbf{PPIN}_n$ specifies that an assignment pattern α is hoisted to the entry of node n either if α is a hoisting candidate in n or α has been hoisted to the exit of node n and is not blocked in n. An assignment pattern α is hoisted to the exit of a node n, if α exists at the entry of all successor nodes.

At the entry/exit of each node n satisfying $\mathbf{PPIN}_n(\alpha)$/ $\mathbf{PPOUT}_n(\alpha)$, a mapping assignment α may be inserted. Of course, whenever a mapping assignment is inserted on a path s, a corresponding redundant assignment shall be eliminated on s. Otherwise, it could happen that the number of mapping assignments is increased. We succeed in eliminating (partially)

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

Local predicates: $N \times \mathcal{D} \rightarrow \{true, false\}$

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

Data flow equations: [1]

$$\mathbf{PPOUT}_n(\alpha) = \begin{cases} \mathit{false} & \mathit{if} \quad n = \mathbf{e} \\ \prod\limits_{m \in succ(n)} \mathbf{PPIN}_m(\alpha) & \mathit{otherwise} \end{cases}$$

$$\mathbf{PPIN}_n(\alpha) = \mathbf{LocHoist}_n(\alpha) + \mathbf{PPOUT}_n(\alpha) * \overline{\mathbf{LocBlock}_n(\alpha)}$$

Insertion Points: [2 3]

$$\mathbf{INSERT}_n(\alpha) \;=_{df}\; \mathbf{PPOUT}^\star_n(\alpha) * \overline{\mathbf{AVOUT}^\star_n(\alpha)} * (\overline{\mathbf{PPIN}^\star_n(\alpha)} + \mathbf{LocBlock}_n(\alpha))$$

$$\mathbf{INSSYNTH}_{mn}(\alpha) \;=_{df}\; \overline{\mathbf{PPOUT}^\star_m(\alpha)} * \mathbf{PPIN}^\star_n(\alpha) * \overline{\mathbf{AVOUT}^\star_m(\alpha)}$$

Elimination Points: [2 4]

$$\mathbf{REDUND}_n(\alpha) \;=_{df}\; \mathbf{LocHoist}_n(\alpha) * \mathbf{PPIN}^\star_n(\alpha)$$

Key:

[1] Boolean conjunctions are denoted by $*$ and Π, Boolean disjunctions by $+$ and Σ, and Boolean negation is denoted by a bar (e.g. $\overline{\mathrm{A}}$).

[2] $\mathbf{PPIN}^\star$, $\mathbf{PPOUT}^\star$, and $\mathbf{AVOUT}^\star$ denote the solutions of the equation systems for hoistability and availability.

[3] Mapping assignment under consideration is inserted at the exit or along the edge of nodes satisfying **INSERT** or **INSSYNTH**.

[4] Mapping assignment under consideration is deleted at nodes satisfying **REDUND**.

Table 4.1: Partial Redundancy Elimination

redundant mapping assignments whenever a mapping assignment can be eliminated on a path without inserting a new instance of that pattern. Insertion and elimination of mapping assignments are controlled by $\mathbf{INSERT}_n$, $\mathbf{INSSYNTH}_{mn}$, and $\mathbf{REDUND}_n$. $\mathbf{INSERT}_n(\alpha)$ and $\mathbf{INSSYNTH}_{mn}(\alpha)$ specify whether mapping assignment α shall be placed at the exit of node n or along the edge (m, n), respectively. Basically, $\mathbf{INSERT}_n$ identifies mapping assignments which are not available at the exit of node n and cannot be hoisted through n, i.e. mapping assignments are hoisted maximally and inserted at the exit of blocking nodes. Edge placement is achieved by $\mathbf{INSSYNTH}_{mn}$: whenever a mapping assignment may be placed at the entry of a node n but not at the exit of a predecessor node m, it is placed along the edge (m, n). Note that the explicit support of edge placement makes edge splitting superfluous. Finally, $\mathbf{REDUND}_n$ denotes mapping assignments which either have already been redundant in node n or have become redundant by newly inserted mapping assignments and can hence be eliminated.

In order to avoid insertion of redundant mapping assignments, $\mathbf{INSERT}_n$ and $\mathbf{INSSYNTH}_{mn}$ rely on the availability of mapping assignments. Table 4.2 displays the data flow equations for availability and partial availability.[1] The global predicates

(Partial) Availability: (Let $\alpha \in \mathcal{D}$)

Local predicates: $N \times \mathcal{D} \rightarrow \{true, false\}$

- $\mathbf{LocAvail}_n(\alpha)$: There is a mapping assignment α in n not followed by some assignment which is a blockade.
- $\mathbf{LocBlockAvail}_n(\alpha)$: The mapping assignment α is blocked by some assignment of n.

Availability:

$$\mathbf{AVIN}_n(\alpha) = \begin{cases} \mathit{false} & \mathit{if} \;\; n = \mathbf{s} \\ \prod\limits_{m \in pred(n)} \mathbf{AVOUT}_m(\alpha) & \mathit{otherwise} \end{cases}$$

$$\mathbf{AVOUT}_n(\alpha) = \mathbf{LocAvail}_n(\alpha) + \mathbf{AVIN}_n(\alpha) * \overline{\mathbf{LocBlockAvail}_n(\alpha)}$$

Partial Availability:

$$\mathbf{PAVIN}_n(\alpha) = \begin{cases} \mathit{false} & \mathit{if} \;\; n = \mathbf{s} \\ \sum\limits_{m \in pred(n)} \mathbf{PAVOUT}_m(\alpha) & \mathit{otherwise} \end{cases}$$

$$\mathbf{PAVOUT}_n(\alpha) = \mathbf{LocAvail}_n(\alpha) + \mathbf{PAVIN}_n(\alpha) * \overline{\mathbf{LocBlockAvail}_n(\alpha)}$$

Table 4.2: Data flow equations for (partial) availability.

$\mathbf{AVIN}_n/\mathbf{AVOUT}_n$ and $\mathbf{PAVIN}_n/\ \mathbf{PAVOUT}_n$ denote mapping assignments which are available and partially available at the entry/exit of a basic block n, respectively. The data flow

[1] Partial availability is used later on.

equations for availability and partial availability are given as usual [38]. The local predicate **LocBlockAvail** differs from **LocBlock** in considering modifications only; e.g. contrary to code motion mapping assignments of the form map(A) := rhs are not blocked by references to variable A in case of the availability problem. **LocAvail** denotes *downwards exposed* mapping assignments: a mapping assignment inside a basic block of the form map(A) := rhs is locally available at the exit, if it is not followed by another mapping assignment with left-hand side A or any definition of a variable used in the rhs-term. Available and partial available mapping assignment analysis differ only in the confluence operator: availability is an all-problem Π whereas partial availability is an any-problem Σ.

Figure 4.4 illustrates that the data flow system specified in Table 4.1 also performs redundant assignment hoistings. The mapping assignment of node **4** is hoisted upwards and

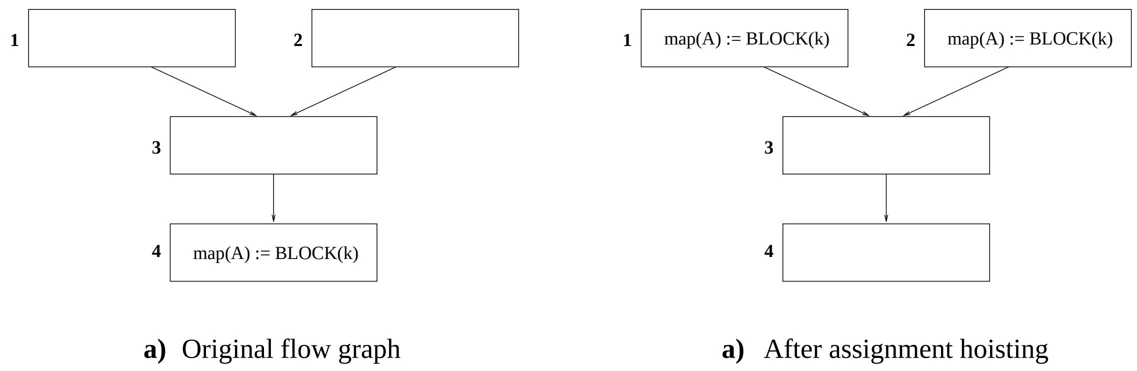


Figure 4.4: Unnecessary assignment hoisting resulting in replication.

replicated along all incoming edges of node **3** without any performance gain as shown in Figure 4.4(b). In order to prevent such redundant code motions, hoisting of mapping assignments is guided by two *profitability terms* which are added to the **PPIN** equation of Table 4.1. The final data flow system is shown in Table 4.3. Obviously, an improvement is only possible if a mapping assignment is partially redundant which is covered by the first profitability term **PAVIN**. Moreover, the second profitability term Σ ensures that there is at least one path where the mapping assignment is redundant or can be moved further, respectively. Note that the second profitability term introduces bidirectional dependences. The profitability terms recognize that hoisting of assignment pattern map(A) := BLOCK(k) of node **4** in Figure 4.4 does not result in any reduction of mapping assignments and the code remains unchanged.

4.3.2 Bidirectional Partial Dead Code Elimination

The idea of partial dead code elimination is to move assignments in the direction of the control flow. Intuitively, this places them in a context as *specific* as possible, and maximizes the potential of dead code which can subsequently be eliminated. This transformation idea is illustrated in Figure 4.5. The assignment x:=a+b at node **2** is partially dead, since it is dead with respect to the assignment at node **4** and live on paths entering node **3** via edge **(2,4)**.

Bidirectional Partial Redundancy Elimination: (Let $\alpha \in \mathcal{D}$)

Local predicates: $N \times \mathcal{D} \rightarrow \{true, false\}$

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

Data flow equations: [1]

$$\mathbf{PPOUT}_n(\alpha) = \begin{cases} \mathit{false} & \mathit{if}\ \ n = \mathbf{e} \\ \prod\limits_{m \in succ(n)} \mathbf{PPIN}_m(\alpha) & \mathit{otherwise} \end{cases}$$

$$\begin{aligned} \mathbf{PPIN}_n(\alpha) = {} & (\mathbf{LocHoist}_n(\alpha) + \mathbf{PPOUT}_n(\alpha) * \overline{\mathbf{LocBlock}_n(\alpha)}) * \mathbf{PAVIN}^\star_n(\alpha) * \\ & \sum_{m \in pred(n)} (\mathbf{PPIN}_m(\alpha) * \overline{\mathbf{LocHoist}_m(\alpha)} + \mathbf{AVOUT}^\star_m(\alpha)) \end{aligned}$$

Insertion Points: [2 3]

$$\mathbf{INSERT}_n(\alpha) =_{df} \mathbf{PPOUT}^\star_n(\alpha) * \overline{\mathbf{AVOUT}^\star_n(\alpha)} * (\overline{\mathbf{PPIN}^\star_n(\alpha)} + \mathbf{LocBlock}_n(\alpha))$$

$$\mathbf{INSSYNTH}_{mn}(\alpha) =_{df} \overline{\mathbf{PPOUT}^\star_m(\alpha)} * \mathbf{PPIN}^\star_n(\alpha) * \overline{\mathbf{AVOUT}^\star_m(\alpha)}$$

Elimination Points: [2 4]

$$\mathbf{REDUND}_n(\alpha) =_{df} \mathbf{LocHoist}_n(\alpha) * \mathbf{PPIN}^\star_n(\alpha)$$

Key:

[1] Boolean conjunctions are denoted by $*$ and Π, Boolean disjunctions by $+$ and Σ, and Boolean negation is denoted by a bar (e.g. $\overline{\mathrm{A}}$).

[2] $\mathbf{PPIN}^\star$, $\mathbf{PPOUT}^\star$, $\mathbf{PAVIN}^\star$, and $\mathbf{AVOUT}^\star$ denote the solutions of the equation systems for hoistability and availability.

[3] Mapping assignment under consideration is inserted at the exit or along the edge of nodes satisfying **INSERT** or **INSSYNTH**.

[4] Mapping assignment under consideration is deleted at nodes satisfying **REDUND**.

Table 4.3: Partial Redundancy Elimination

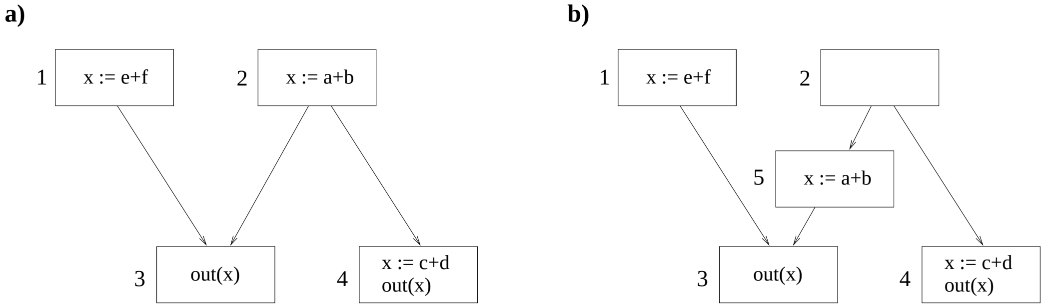


Figure 4.5: Sinking and edge placement.

Sinking the assignment x:=a+b from node **2** to node **3** would eliminate the partial deadness. However, insertion of the assignment at node **3** is not allowed, since on paths entering node **3** via edge **(1,3)** the assignment x:=e+f would be overwritten. But the partial deadness can be eliminated by inserting a synthetic node on critical edge **(1,3)** as shown in Figure 4.5(b).

Analogously to assignment hoisting, let us discuss for assignment sinking the safety issues first. The insertion of a mapping assignment α at a program point p is safe if on each path ending in p, assignment α is executed and between the last execution of α and p neither one of the operands is changed nor the left-hand side of α changed or used. Table 4.4 displays the data flow system for assignment sinking. The data flow equations are based on the local predicates **LocSink** and **LocBlock**. $\mathbf{LocSink}_n(\alpha)$ specifies whether there is a *sinking candidate* of α inside basic block n. Sinking candidates are mapping assignments inside basic blocks which are not blocked by a subsequent instruction of the same basic block (such mapping assignments are also called *downwards exposed*). The rules for blocking are for hoisting and sinking of assignment patterns the same. Hence, the local predicate **LocBlock** is identically defined to assignment hoisting (cf. Section 4.3.1). $\mathbf{LocBlock}_n(\alpha)$ specifies whether assignment pattern α is blocked by instructions inside basic block n and, hence, sinking through node n is prevented.

The **PPIN/PPOUT** equations assure the safety of code sinking. $\mathbf{PPIN}_n$ specifies that an assignment pattern α is moved to the exit of node n either if α is a sinking candidate in n or α has been moved to the entry of node n and is not blocked in n. An assignment pattern α is moved to the entry of a node n, if α exists at the exit of all predecessor nodes.

At the entry/exit of each node n satisfying $\mathbf{PPIN}_n(\alpha)$/ $\mathbf{PPOUT}_n(\alpha)$, a mapping assignment α may be inserted. Of course, whenever a mapping assignment is inserted on a path s, a corresponding dead assignment shall be eliminated on s. Otherwise, it could happen that the number of mapping assignments is increased. We succeed in eliminating (partially) dead mapping assignments whenever a mapping assignment can be eliminated on a path without inserting a new instance of that pattern. Insertion and elimination of mapping assignments are controlled by $\mathbf{INSERT}_n$, $\mathbf{INSSYNTH}_{mn}$, and $\mathbf{DEAD}_n$. $\mathbf{INSERT}_n(\alpha)$ and $\mathbf{INSSYNTH}_{mn}(\alpha)$ specify whether mapping assignment α shall be placed at the entry of node n or along the edge (m,n), respectively. Basically, $\mathbf{INSERT}_n$ identifies mapping assignments which are not dead at the entry of node n and cannot be moved further through n, i.e.

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

Local predicates: $N \times \mathcal{D} \rightarrow \{true, false\}$

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

Data flow equations: [1]

$$\mathbf{PPIN}_n(\alpha) = \begin{cases} \mathit{false} & \mathit{if} \quad n = \mathbf{s} \\ \prod_{m \in pred(n)} \mathbf{PPOUT}_m(\alpha) & \mathit{otherwise} \end{cases}$$

$$\mathbf{PPOUT}_n(\alpha) = \mathbf{LocSink}_n(\alpha) + \mathbf{PPIN}_n(\alpha) * \overline{\mathbf{LocBlock}_n(\alpha)}$$

Insertion Points: [2 3]

$$\mathbf{INSERT}_n(\alpha) =_{df} \mathbf{PPIN}^\star_n(\alpha) * \overline{\mathbf{DDIN}^\star_n(\alpha)} * (\overline{\mathbf{PPOUT}^\star_n(\alpha)} + \mathbf{LocBlock}_n(\alpha))$$

$$\mathbf{INSSYNTH}_{mn}(\alpha) =_{df} \overline{\mathbf{PPIN}^\star_n(\alpha)} * \mathbf{PPOUT}^\star_m(\alpha) * \overline{\mathbf{DDIN}^\star_n(\alpha)}$$

Elimination Points: [2 4]

$$\mathbf{DEAD}_n(\alpha) =_{df} \mathbf{LocSink}_n(\alpha) * \mathbf{PPOUT}_n(\alpha)$$

Key:

[1] Boolean conjunctions are denoted by $*$ and Π, Boolean disjunctions by $+$ and Σ, and Boolean negation is denoted by a bar (e.g. $\overline{\mathrm{A}}$).

[2] $\mathbf{PPIN}^\star$, $\mathbf{PPOUT}^\star$, and $\mathbf{DDIN}^\star$ denote the solutions of the equation systems for sinkability and deadness.

[3] Mapping assignment under consideration is inserted at the exit or along the edge of nodes satisfying **INSERT** or **INSSYNTH**.

[4] Mapping assignment under consideration is deleted at nodes satisfying **DEAD**.

Table 4.4: Partial Dead Code Elimination

mapping assignments are moved maximally and inserted at the entry of blocking nodes. Edge placement is achieved by $\mathbf{INSSYNTH}_{mn}$: whenever a mapping assignment may be placed at the exit of a node m but not at the entry of a successor node n, it is placed along the edge (m, n). Note that the explicit support of edge placement makes edge splitting superfluous. Finally, $\mathbf{DEAD}_n$ denotes mapping assignments which either have already been dead in node n or have become dead by newly inserted mapping assignments and can hence be eliminated.

In order to avoid insertion of dead mapping assignments, $\mathbf{INSERT}_n$ and $\mathbf{INSSYNTH}_{mn}$ rely on deadness information. Table 4.5 displays the data flow equations for deadness and partial deadness.[2] The global predicates $\mathbf{DDI}_n/\mathbf{DDO}_n$ and $\mathbf{PDDI}_n/\mathbf{PDDO}_n$ denote variables

---

(Partial) Deadness of Mappings: (Let $\alpha \in \mathcal{D}$, Var denote the set of data objects declared in the program, and $v \in Var$)

Local predicates: $N \times Var \rightarrow \{true, false\}$

- $\mathbf{LocMod}_n(v)$: There is a change of the mapping of v prior to any use of that mapping in n.
- $\mathbf{LocUsed}_n(v)$: There is a use of the mapping of v in n.

Dead Mapping Analysis:

$$\mathbf{DDO}_n(v) = \begin{cases} false & if \;\; n = \mathbf{e} \\ \prod\limits_{m \in succ(n)} \mathbf{DDI}_m(v) & otherwise \end{cases}$$

$$\mathbf{DDI}_n(v) = \mathbf{LocMod}_n(v) + \mathbf{DDO}_n(v) * \overline{\mathbf{LocUsed}_n(v)}$$

Partial Dead Mapping Analysis:

$$\mathbf{PDDO}_n(v) = \begin{cases} false & if \;\; n = \mathbf{e} \\ \sum\limits_{m \in succ(n)} \mathbf{PDDI}_m(v) & otherwise \end{cases}$$

$$\mathbf{PDDI}_n(v) = \mathbf{LocMod}_n(v) + \mathbf{PDDO}_n(v) * \overline{\mathbf{LocUsed}_n(v)}$$

Dead and Partial Dead Mapping Assignments: (Let $\alpha \equiv map(v) := rhs$)

$$\begin{array}{lcl} \mathbf{DDIN}^{\star}_n(\alpha) \text{ resp. } \mathbf{DDOUT}^{\star}_n(\alpha) & =_{df} & \mathbf{DDI}^{\star}_n(v) \text{ resp. } \mathbf{DDO}^{\star}_n(v) \\ \mathbf{PDDIN}^{\star}_n(\alpha) \text{ resp. } \mathbf{PDDOUT}^{\star}_n(\alpha) & =_{df} & \mathbf{PDDI}^{\star}_n(v) \text{ resp. } \mathbf{PDDO}^{\star}_n(v) \end{array}$$

Table 4.5: Data flow system for (partial) dead mapping assignments.

---

whose mappings are dead and partially dead on the entry/exit of a basic block n, respectively. The data flow equations for deadness and partial deadness of mappings are given as usual [38]. The local predicate $\mathbf{LocMod}_n(v)$ specifies whether the mapping of array v is dead in basic block n. $\mathbf{LocUsed}_n(v)$ denotes whether the mapping of array v is used in basic block n.

[2] Partial deadness is used later on.

$\mathbf{DDI}^\star$, $\mathbf{DDO}^\star$, $\mathbf{PDDI}^\star$, and $\mathbf{PDDO}^\star$ denote the solutions for (partial) deadness. According to our definition, a mapping assignment is dead if the mapping of the corresponding variable is dead. Partial dead distribution analysis differs from dead distribution analysis only in the confluence operator which is changed from Π to Σ.

Figure 4.6 illustrates that the data flow system specified in Table 4.4 also performs redundant assignment sinkings. The mapping assignment of node **1** is moved downwards and

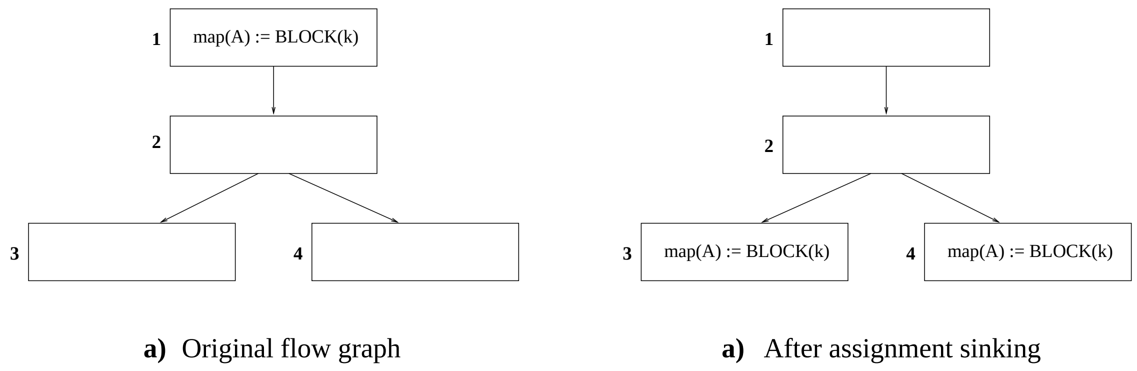


Figure 4.6: Unnecessary assignment sinking resulting in replication.

replicated along all outgoing edges of node **2** without any performance gain as shown in Figure 4.6(b). In order to prevent such redundant code motions, hoisting of mapping assignments is guided by two *profitability terms* which are added to the **PPIN** equation of Table 4.4. The final data flow system is shown in Table 4.6. Obviously, no performance gain is possible for totally live mapping assignments. Thus the first profitability term **PDDOUT** requires that the mapping is partially dead. The second profitability term Σ ensures that there is at least one path where the assignment is dead or can be moved further, respectively. Note that the second profitability term introduces bidirectional dependences. In Figure 4.6 the profitability terms recognize that sinking of assignment pattern $\mathsf{map(A)} := \mathsf{BLOCK(k)}$ of node **1** does not result in any reduction of mapping assignments and the code remains unchanged.

Similar to bidirectional partial redundancy elimination systems, we must prove for bidirectional partial dead code elimination that the transformations are safe and profitable. It has to be shown that for all blocks i with $dead_i = true$, mapping assignments satisfying $\mathbf{LocSink}_i = true$ are dead, and that on any program path the number of mapping assignments has not been increased.

Lemma 4.3.1.
After insertion of mapping assignments at the entry of all nodes with $\mathbf{INSERT} = true$ *and in all synthetic nodes with* $\mathbf{INSSYNTH} = true$ *respectively, the new value of* $\mathbf{DDOUT}_k$ *for any node* k *with* $\mathbf{PPOUT}_k = true$ *will be* $\mathbf{DDOUT}'_k = true$.

Bidirectional Partial Dead Code Elimination: (Let $\alpha \in \mathcal{D}$)

Local predicates: $N \times \mathcal{D} \rightarrow \{true, false\}$

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

Data flow equations: [1]

$$\mathbf{PPIN}_n(\alpha) = \begin{cases} \mathit{false} & \mathit{if} \;\; n = \mathbf{s} \\ \prod_{m \in pred(n)} \mathbf{PPOUT}_m(\alpha) & \mathit{otherwise} \end{cases}$$

$$\mathbf{PPOUT}_n(\alpha) = (\mathbf{LocSink}_n(\alpha) + \mathbf{PPIN}_n(\alpha) * \overline{\mathbf{LocBlock}_n(\alpha)}) * \mathbf{PDDOUT}^{\star}_n(\alpha) * \sum_{m \in succ(n)} (\mathbf{PPOUT}_m(\alpha) * \overline{\mathbf{LocSink}_m(\alpha)} + \mathbf{DDIN}^{\star}_m(\alpha))$$

Insertion Points: [2 3]

$$\mathbf{INSERT}_n(\alpha) =_{df} \mathbf{PPIN}^{\star}_n(\alpha) * \overline{\mathbf{DDIN}^{\star}_n(\alpha)} * (\overline{\mathbf{PPOUT}^{\star}_n(\alpha)} + \mathbf{LocBlock}_n(\alpha))$$

$$\mathbf{INSSYNTH}_{mn}(\alpha) =_{df} \overline{\mathbf{PPIN}^{\star}_n(\alpha)} * \mathbf{PPOUT}^{\star}_m(\alpha) * \overline{\mathbf{DDIN}^{\star}_n(\alpha)}$$

Elimination Points: [2 4]

$$\mathbf{DEAD}_n(\alpha) =_{df} \mathbf{LocSink}_n(\alpha) * \mathbf{PPOUT}_n(\alpha)$$

Key:

[1] Boolean conjunctions are denoted by $*$ and Π, Boolean disjunctions by $+$ and Σ, and Boolean negation is denoted by a bar (e.g. $\overline{\text{A}}$).

[2] $\mathbf{PPIN}^{\star}$, $\mathbf{PPOUT}^{\star}$, $\mathbf{PDDOUT}^{\star}$, and $\mathbf{DDIN}^{\star}$ denote the solutions of the equation systems for sinkability and deadness.

[3] Mapping assignment under consideration is inserted at the exit or along the edge of nodes satisfying **INSERT** or **INSSYNTH**.

[4] Mapping assignment under consideration is deleted at nodes satisfying **DEAD**.

Table 4.6: Partial Dead Code Elimination

Proof.

Let us assume an arbitrary but fixed path $p \in \mathbf{P}[k, i]$, denoted by $[k, i]$, with $\mathbf{PPOUT}_k =$ *true* and $\mathbf{DDOUT}'_k =$ *false*. Then there must exist such a node i such that $\mathbf{DDIN}_i =$ *false*, $\mathbf{INSERT}_i =$ *false*, $\mathbf{LocBlock}_i =$ *true*, and the path $]k, i[$ is an empty path with $\mathbf{INSERT}_u =$ *false*, $u \in]k, i[$, and $\mathbf{INSSYNTH}_{wv} =$ *false*, $v \in]k, i], w \in [k, i[$, respectively. The first condition expresses that the mapping assignment has not already been dead whereas the second condition assures that no new mapping assignment has been inserted which would result in dead code. Then $\mathbf{DDIN}'_i =$ *false* which is propagated by the confluence operator from i to k till $\mathbf{DDOUT}'_k =$ *false*.

Since $\mathbf{INSERT}_i =$ *false*, $\mathbf{DDIN}_i =$ *false*, and $\mathbf{LocBlock}_i =$ *true*, we obtain $\mathbf{PPIN}_i =$ *false* by the **INSERT** equation. Let j be the predecessor of i in $]k, i[$. Then $\mathbf{INSSYNTH}_{ji} =$ *false* with $\mathbf{PPIN}_i =$ *false* and $\mathbf{DDIN}_i =$ *false* implies $\mathbf{PPOUT}_j =$ *false* by the **INSSYNTH** equation. If

(i). $j = k$ we have a contradiction on $\mathbf{PPOUT}_k$ which concludes the proof.

(ii). Otherwise $\mathbf{INSERT}_j =$ *false* with $\mathbf{DDIN}_j =$ *false* [3], and $\mathbf{PPOUT}_j =$ *false* implies $\mathbf{PPIN}_j =$ *false* by the **INSERT** equation.

Since we have now for node j the same situation like for node i, we can proceed with the predecessor node of node j in the same way until condition (i) is fulfilled.

■

Theorem 4.3.1 (Dead Mapping Assignments).
After insertion of mapping assignments at the entry of all nodes with **INSERT** $=$ *true and in all synthetic nodes with* **INSSYNTH** $=$ *true respectively, any mapping assignment satisfying* $\mathbf{LocSink}_i =$ *true is dead in all nodes i with* $\mathbf{PPOUT}_i =$ *true.*

Proof.

It is easy to see that $\mathbf{LocSink}_i * \mathbf{PPOUT}_i =$ *true* leads to $\mathbf{PPOUT}_i =$ *true*. By Lemma 4.3.1, $\mathbf{DDOUT}'_i =$ *true*. Since $\mathbf{LocSink}_i$ cannot be changed by insertions, $\mathbf{LocSink}_i * \mathbf{DDOUT}'_i =$ *true* which indicates dead code. ■

Lemma 4.3.2.
Every path ending at the entry of node i such that $\mathbf{INSERT}_i =$ *true or* $\mathbf{INSSYNTH}_{ji} =$ *true, contains a dead mapping assignment.*

Proof.

On the one hand, $\mathbf{INSERT}_i =$ *true* implies $\mathbf{PPIN}_i =$ *true* and $\mathbf{AVOUT}_j =$ *true*, $j \in pred(i)$. On the other hand, $\mathbf{INSSYNTH}_{ji} =$ *true* implies $\mathbf{PPOUT}_j =$ *true* and $\mathbf{AVOUT}_j =$ *true*. Hence, there exists an assignment on every path ending at node i. We have to show now that these assignments are dead.

[3] $\mathbf{DDIN}_i =$ *false* implies $\mathbf{DDOUT}_j =$ *false* which leads to $\mathbf{DDIN}_j =$ *false* since $]k, i[$ is an empty path.

Assume a node k which contains such an assignment which is not dead and assume a fixed path $p \in \mathbf{P}[k,i]$, denoted by $[k,i]$. By Theorem 4.3.1, $\mathbf{LocSink}_k * \mathbf{PPOUT}_k = \mathit{false}$ with $]k,i[$ empty path. This equation leads to $\mathbf{PPOUT}_k = \mathit{false}$ which implies $\mathbf{PPIN}_l = \mathit{false}$ with $l \in succ(k)$ and $l \in]k,i[$. If

(i). $l = i$ we have a contradiction on $\mathbf{PPIN}_i$ or $\mathbf{PPOUT}_j$ respectively which concludes the proof.

(ii). Otherwise $\mathbf{PPIN}_l = \mathit{false}$, $\mathbf{LocBlock}_l = \mathit{false}$, and $\mathbf{LocSink}_l = \mathit{false}$ leads to $\mathbf{PPOUT}_l = \mathit{false}$.

For node l we have the same situation like for node k. Hence, the same reasoning can be applied for $]k,i[$ until condition (i) applies.

∎

Lemma 4.3.3.
Any path starting from the exit of a node k with $dead_k = true$ and ending in a node i such that $\mathbf{INSERT}_i = true$ or $\mathbf{INSSYNTH}_{ji} = true$, cannot contain another node with either $\mathbf{INSERT} = true$ or $\mathbf{INSSYNTH} = true$.

Proof.

Assume a fixed path $p \in \mathbf{P}[k,i]$, denoted by $[k,i]$. By Lemma 4.3.2, $\mathbf{LocSink}_k * \mathbf{PPOUT}_k = \mathit{true}$ with $]k,i[$ empty path. Assume that there exists an $u \in]k,i[$ with $\mathbf{INSERT}_u = \mathit{true}$ or $\mathbf{INSSYNTH}_{vu} = \mathit{true}$. From $\mathbf{INSERT}_u = \mathit{true}$ we get $\mathbf{PPOUT}_u = \mathit{false}$. $\mathbf{INSSYNTH}_{vu} = \mathit{true}$ implies $\mathbf{PPIN}_u = \mathit{false}$ and since $\mathbf{LocSink}_u = \mathit{false}$ we get $\mathbf{PPOUT}_u = \mathit{false}$.

As we have seen in the proof of Lemma 4.3.2, $\mathbf{PPOUT}_u = \mathit{false}$ propagates through $]u,i[$ until $\mathbf{PPOUT}_j = \mathit{false}$, $j \in]u,i[$, $j \in pred(i)$, and finally $\mathbf{PPIN}^S_i = \mathit{false}$. However, $\mathbf{INSERT}_i = \mathit{false}$ or $\mathbf{INSSYNTH}_{ji} = \mathit{false}$ is a contradiction which concludes the proof. ∎

Theorem 4.3.2 (Profitability).
The bidirectional partial dead code elimination algorithm does not increase the number of mapping assignments along any program path.

Proof.

Lemma 4.3.2 and Lemma 4.3.3 imply Theorem 4.3.2. ∎

4.3.3 Bidirectional Mapping Assignment Placement Framework

The mapping placement framework consists of eliminating partially dead mapping assignments followed by eliminating partially redundant mapping assignments. Thus, the resulting combined algorithm covers following optimizations: dead code elimination, suppression of partial dead code, redundancy elimination, suppression of partial redundancies, and elimination of loop-invariant mapping assignments by hoisting or sinking the assignments just before or just after the loop, respectively. The safety constraints which control the motion of mapping assignments have been chosen for both partial dead code elimination and partial redundancy elimination such that it is guaranteed that the actual mapping for any array reference will be identical at runtime for the unoptimized and optimized version of the program. As shown in our examples, partial dead code elimination must be performed prior to partial redundancy elimination in order to succeed to move mapping assignments out of loops in such typical situations like procedure calls with remappings at the entry and exit.

Algorithm 4.3.1 : Mapping Placement

A. Solve bidirectional partial dead code elimination system

(i) for each node n: compute local sets $\mathbf{LocSink}_n$, $\mathbf{LocBlock}_n$, $\mathbf{LocMod}_n$, and $\mathbf{LocUsed}_n$

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

(iii) resolution of the bidirectional system of equations yielding $\mathbf{PPIN}_n/\mathbf{PPOUT}_n$ for each n

(iv) for each node n: for each $\mathbf{INSERT}_n$ and $\mathbf{INSSYNTH}_{mn}$ insert a mapping assignment at the entry of node n or along the edge (m, n), respectively

(v) for each node n: for each $\mathbf{DEAD}_n$ delete last occurrence of the mapping assignment in n

B. Solve bidirectional partial redundancy elimination system

(i) for each node n: compute local sets $\mathbf{LocHoist}_n$, $\mathbf{LocBlock}_n$, $\mathbf{LocAvail}_n$, and $\mathbf{LocBlockAvail}_n$

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

(iii) resolution of the bidirectional system of equations yielding $\mathbf{PPIN}_n/\mathbf{PPOUT}_n$ for each n

(iv) for each node n: for each $\mathbf{INSERT}_n$ and $\mathbf{INSSYNTH}_{mn}$ insert a mapping assignment at the exit of node n or along the edge (m, n), respectively

(v) for each node n: for each $\mathbf{REDUND}_n$ delete first occurrence of the mapping assignment in n

Figure 4.7: Mapping Placement Framework

The distribution placement framework is presented in Figure 4.7. The unidirectional flow systems (**DDIN/DDOUT**, **PDDIN/PDDOUT**, **AVIN/AVOUT**, and **PAVIN/PAVOUT**) can be solved easily using existing techniques (e.g. monotone data flow systems [2]). It has been shown that for reducible flow graphs the iterative approach [2] takes at most $d + 2$ iterations with d denoting the loop-connectedness of the graph [41]. Since in practice d is often 3 or less [58], the algorithm will terminate on the average after at most 5 iterations. But note that dead code analysis and redundancy analysis, on which bidirectional partial dead code elimination and partial redundancy elimination rely, are mandatory analyses for optimizing compilers in any way such that both can be considered in part for free. Since interval-based *elimination methods* are not directly applicable to bidirectional flow systems, usually the iterative approach is used. But classical monotone data flow frameworks which assume dependences only in one direction cannot be applied directly as well. In [19] the resolution of the bidirectional data flow problems has been discussed. There it is proposed to split bidirectional equations into a sequence of unidirectional ones which makes classical solution methods and complexity bounds amenable for bidirectional problems.

4.3.4 Deficiencies of the Bidirectional Approach

Originally bidirectional partial redundancy elimination has been developed in the context of subexpression elimination [65]. In that context it was important to avoid unnecessary code motion which has no impact on performance but can result in superfluous register pressure when expression evaluations are separated from their uses. Hence it was an issue to be *minimal invasive*: let computations remain at their original program place and perform code motion only in case of possible performance gains as expressed by the profitability terms. However profitability is not that simple. The application of the same strategy to assignments may hurt as shown in Figure 4.8. Node **1** contains a mapping assignment which assigns a block

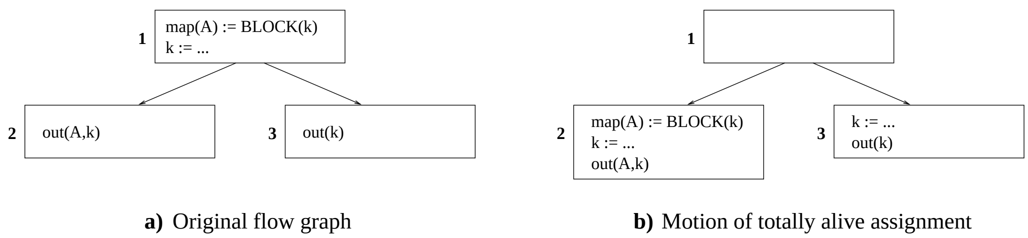


Figure 4.8: Partially dead mapping assignment not removed.

distribution with block size k to array A followed by an ordinary assignment to variable k. The mapping assignment is partially dead, since it is alive on path (1,2) and dead on path (1,3). Sinking of the mapping assignment, however, is blocked by the ordinary assignment. Hence, the partially dead mapping assignment can only be eliminated, if the ordinary assignment is first moved to nodes **2** and **3** which allows as a second-order effect sinking of the mapping assignment as well resulting in the flow graph depicted in Figure 4.8(b). But as the ordinary assignment is totally alive, no performance gains are expected and code motion is prevented by the profitability terms. In this way the original flow graph 4.8(a) remains unchanged and the partial deadness of the mapping assignment could not be removed. Note that we used here for the first time dependences between mapping assignments and ordinary assignments. In the following section we will discuss this issue in more detail.

Thus the attempt to be minimal invasive may hurt in case of second-order effects. Moreover, this minimal-invasive strategy based on the profitability terms as described in our data flow systems are only heuristics as shown in Figure 4.9. The assignment of mapping BLOCK(i) to array A is partially dead in node **1**, since it is alive on the path from node **1** to node **6** and

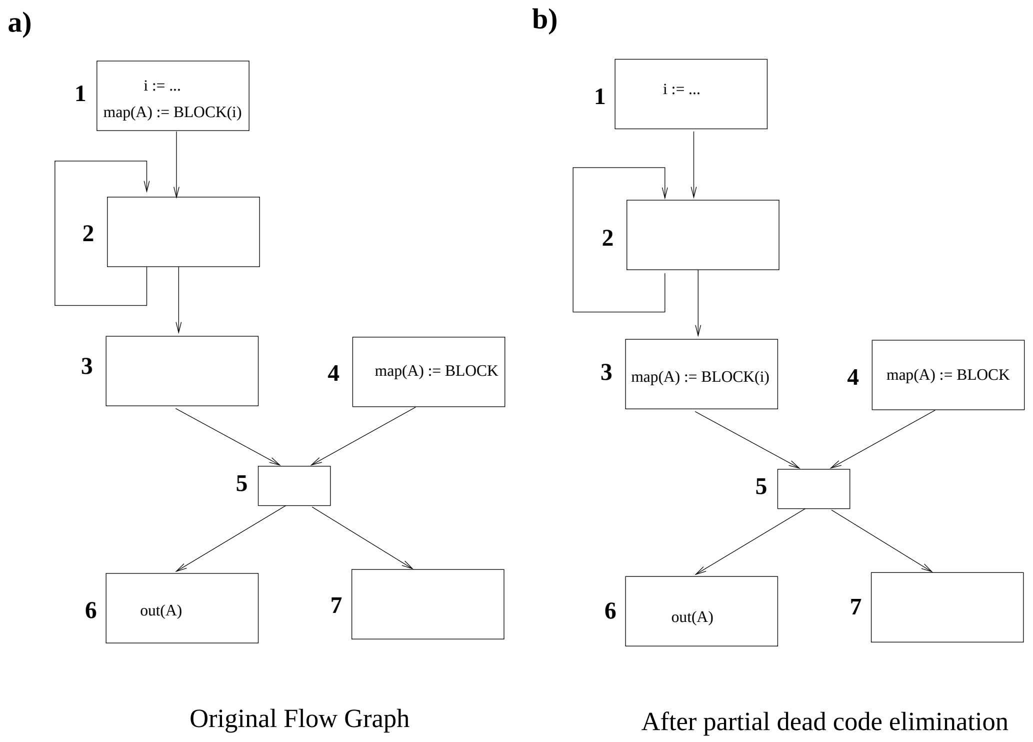


Figure 4.9: Motion strategy not minimal invasive.

dead on the path from node **1** to node **7**. Because of the joining node **5**, the partial deadness of the mapping assignment of node **1** cannot be eliminated and any code motion is unnecessary. Nevertheless, the profitability terms cannot prevent the sinking of the mapping assignment to node **3** as shown in Figure 4.9(b). The reason for that unnecessary code motion is the loop at node **2**. Hence, the profitability terms restrict only heuristically code motion. In case of bidirectional partial redundancy elimination, mapping assignments which are moved are inserted heuristically between the earliest possible program points and the predecessor nodes of the current position. Analogously for bidirectional partial dead code elimination, mapping assignments which are moved are inserted heuristically between the successor nodes of the current position and the latest possible program points.

From the compilation point of view it is an issue to be minimal invasive, i.e. to restructure code only when it is worthwhile. But also complex profitability terms could not achieve this goal. On the other hand, mapping assignments can be very expensive. Hence our goal is to find "optimal" solutions. Intuitively, an "optimal" solution is a program version where the number of executed remappings cannot be reduced further by moving and eliminating mapping assignments. The bidirectional approach tries to prevent unnecessary code motions at the price of transformational power and may miss in this way opportunities to eliminate partially dead or redundant code.

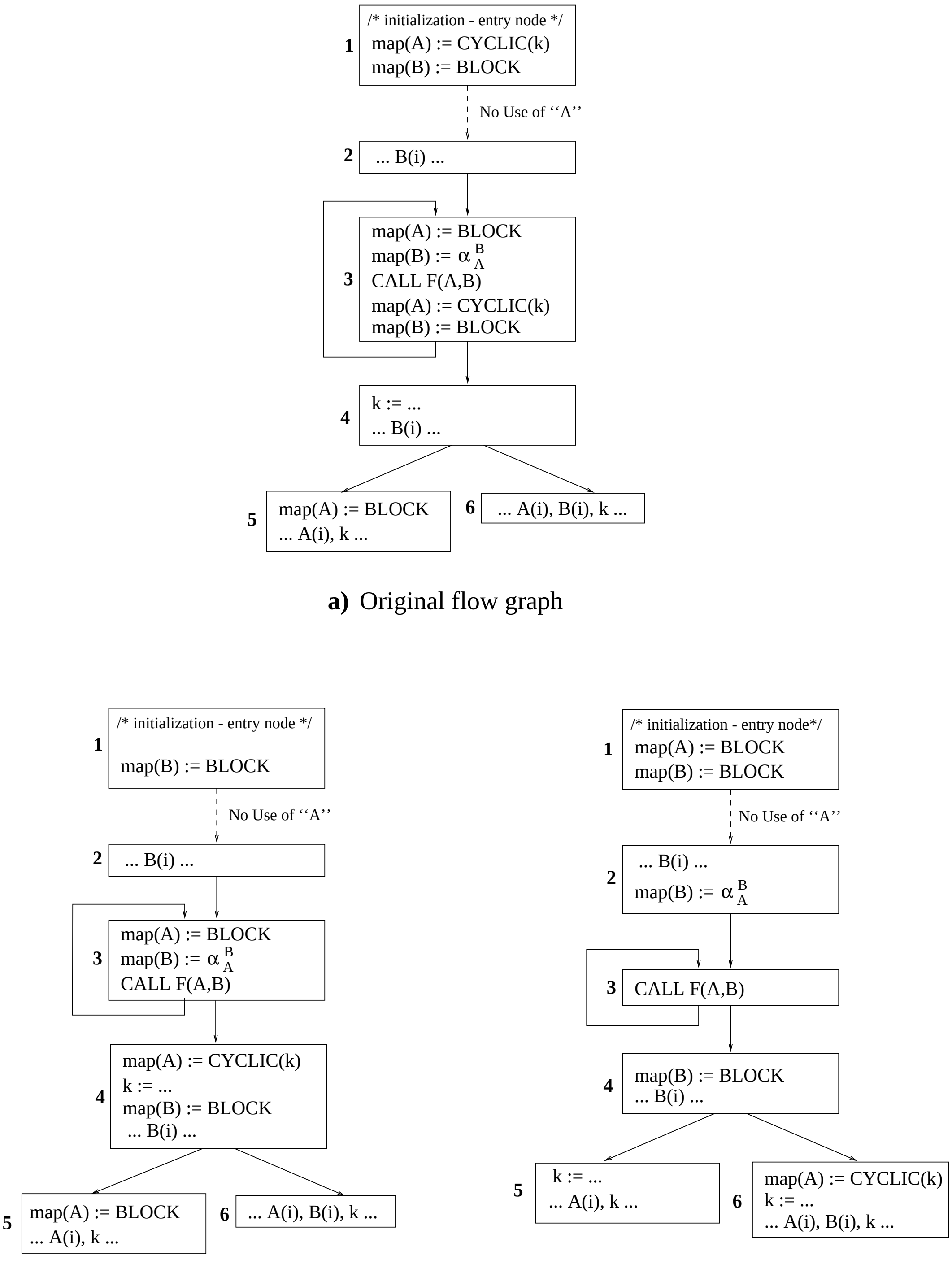
4.4 The MAP-Approach

We have seen that mapping assignments can be very expensive and that motion restrictions for being minimal invasive may prevent beneficial optimizations. In this section we give up motion restrictions and try to find best solutions (cf. [51, 50]). First, we have to investigate whether there exists an "optimal" solution for our problem or not. Intuitively, an "optimal" solution is a program version obtained by code motion and elimination of useless code with the property that the number of mapping assignments executed on each program path is less or equal to that of any other program version under the condition that the branching structure or the semantics of the program is not changed. Second, if such "optimal" programs exist, algorithms are required which are capable of finding those solutions. Important in finding best solutions is the proper handling of second-order effects. In Figure 4.8 we have seen that second-order effects may not only exist between mapping assignments but may occur between mapping assignments and ordinary assignments, as well. The following example illustrates several second-order effects which have to be dealt with in order to obtain "optimal" results.

4.4.1 Second Motivating Example

The example of Figure 4.10 illustrates the full complexity of resolving second-order effects between different statement patterns. The use of alignments for specifying the mapping of arrays introduces dependences between mapping assignments. Parameters in distribution specifications like in BLOCK(i) or CYCLIC(i) introduce dependencies between ordinary and mapping assignments, as well. Both kinds of dependencies introduce in general second-order effects as illustrated in Figure 4.10(a), 4.10(b), and 4.10(c). As before, a subroutine which is transparent for the mappings 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 4.2, the mapping 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 4.10(b)). Whereas the mapping assignment map(B) := BLOCK at node **4** is now totally live, map(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 4.10(b)). Hence, moving the ordinary assignment to k first suspends this blockade and subsequently allows to eliminate the partially dead assignment map(A) := CYCLIC(k) as desired. As a side-effect this turns the mapping assignment at node **5** to be totally redundant, which can thus be eliminated as well. As another second-order effect, the mapping 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 map(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 4.10(c) shows the result of the complete transformation. Note that the optimized flow graph is optimal: it is free of any partially dead and partially redundant mapping assignment.



b) After the PDCE-step

c) Optimized flow graph

Figure 4.10: Complex example: Illustrating the impact of second-order effects.

4.4.2 Fundamentals of the MAP-Approach

In the following we develop the MAP-framework, which, in essence, works by eliminating *partially dead* and *partially redundant* mapping changes whereby contrary to the bidirectional approach our main objective is to deal with second-order effects properly. Besides the second-order effects introduced by interdependences of different statement patterns, which have been described in our examples and can be overcome by repeated applications of the elementary transformations, we will see that interleaving of partial dead code elimination and partial redundancy elimination reveals additional interdependences of the transformations themselves making optimality more intricate as the result depends on the particular sequence of the transformations. Further, 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* MAP focuses on mapping assignments and resolves second-order effects among them, the algorithm of *Full* MAP resolves even all second-order effects between ordinary and mapping 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.

We stepwise develop an hierarchy of MAP-algorithms starting with the algorithms of *Pure* MAP and *Full* MAP, which constitute the kernel of our hierarchy of user-customized algorithms for eliminating unnecessary overhead due to mapping changes. In essence, the hierarchy is based on the trade-off between efficiency and power of the transformation.

As already mentioned, MAP relies on the combined effects of

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

This is important because both subproblems can be solved optimally as it was discussed in [53] and [54] 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 originally designed 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 MAP-algorithms.

4.4.3 Partially Dead Code Elimination (PDCE)

Like MAP, 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. 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 4.7 summarizes the data flow analyses for assignment sinkings and dead assignment eliminations, which uniformly capture ordinary and mapping assignments. In the following we discuss the intuition underlying both analyses.

Similarly to the bidirectional approach, the AS-analysis is controled by the local predicates **LocBlock** and **LocSink** being the handle for restricting assignment sinkings to *admissible* ones. Both predicates cover ordinary and mapping assignments. **LocBlock** prevents motion of ordinary assignments across instructions which use or modify their left-hand side variables, or

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.

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

Insertion Points:

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

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

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

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

Elimination Points: $\mathbf{ELIM}_\iota =_{df} \mathbf{X\text{-}DEAD}^\star_\iota$

Table 4.7: Assignment Sinking and Dead Variable Analysis

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 (the interprocedural case is discussed in the next chapter). In essence, the same blocking constraints we impose on mapping assignments. A mapping assignment of the form map(A) := *rhs* is blocked by any occurrence of A and any definition of a variable used in *rhs*. Analogously to the bidirectional case, this guarantees that assignment sinkings respect the mapping proposed by the programmer: for each array reference the mappings 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 mappings of arrays do not affect the program semantics. In our approach, this can easily be achieved by defining the predicate **LocBlock** for ordinary and mapping 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 mapping assignments inside a basic block which are not blocked by a subsequent assignment of the same basic block (called *downwards exposed*). Again, **N-SINK***, **X-SINK***, and **X-DEAD*** denote the greatest solutions of the equation systems for sinkability and deadness, respectively. All sinking candidates of an assignment pattern under consideration are replaced by instances of this pattern inserted at program points satisfying **N-INSERT** or **X-INSERT**.

Elimination of dead assignments is based on dead variable analysis which is controlled by the local predicates **Used** and **Mod**. For the ease of presentation the equations are specified at instruction level. A variable x is dead at a program point if on every program continuation every use of the value or mapping of x (expressed by **Used**) is preceded by a modification of the value or mapping of x (expressed by **Mod**). The elimination points identify program positions where the value or mapping of variable x is dead and where, hence, ordinary or mapping assignments with left-hand side variable x can be eliminated at those program positions.

The PDCE-Algorithm and its Optimality

Following [53] the second-order effects induced by interdependences of different assignment patterns (cf. Section 4.2) are fully captured by repeatedly applying the elementary transformations of PDCE until the program stabilizes. This is conveniently 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 $\mathcal{P}$. The definition of optimality depends on the set of programs which we consider and on the relation "better". Let

$$\mathcal{G}_{pdce} =_{df} \{G' \mid G \vdash^*_{(\mathsf{AS+DCE})} G'\}$$

denote the *program universe* derivable from G by means of sequences of admissible assignment sinkings and dead code eliminations.

Definition 4.4.1 Relation "better"
A program $G' \in \mathcal{G}_{pdce}$ is better *than a program $G'' \in \mathcal{G}_{pdce}$ iff for* each *assignment pattern the number of assignments executed on each path in G' is less or equal to that in G''.* ∎

Note that the relation "better" does not simply count assignments independently of the assignment pattern, but counts for each assignment pattern the number of assignments. This relation specifies a very strict condition which allows to define optimality without taking execution costs of individual assignments into account.

As proved in [31] the program resulting from PDCE is *best* (*optimal*): it is *better* (in the strict sense as defined above) than any other program in the program set $\mathcal{G}_{pdce}$. In particular, the program finally resulting from PDCE does not depend on the specific order of the elementary transformations: it is uniquely determined up to (irrelevant) local reorderings in basic blocks (cf. [31, 53]).

4.4.4 Partially Redundant Assignment Elimination (PRAE)

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 and the analogy to bidirectional partial redundancy elimination, we only present the data flow equations in Table 4.8 for AH and RAE and do not discuss them in detail here. We just remark that the local predicate **LocBlock** for assignment hoisting is defined

---

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:

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

Redundant Assignment Analysis (RAE): (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 $\alpha \equiv map(A) := ...$).

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

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

Table 4.8: Assignment Hoisting and Redundant Assignment Analysis

---

like its counterpart for assignment sinking. Similarly, this also holds for the local predicate

LocHoist identifying *hoisting candidates*: these are occurrences of ordinary and mapping assignments inside a basic block which are not blocked by a preceding instruction of the same basic block (called *upwards exposed*). Again, **N-HOIST***, **X-HOIST***, and **N-RED*** denote the greatest solution of the equation systems for hoistability and redundancy, respectively. All hoisting candidates of an assignment pattern under consideration are replaced by instances of this pattern inserted at program points satisfying **N-INSERT** or **X-INSERT**. For the ease of presentation redundant assignment analysis is employed at the instruction level. The elimination points identify program positions where an assignment pattern is available allowing to eliminate instances of this pattern at those points.

The PRAE-Algorithm and its Optimality

Like for PDCE, also for PRAE repeated applications of assignment hoistings and redundant assignment eliminations suffice to capture the second-order effects between different statement patterns completely, 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

$$\mathcal{G}_{prae} =_{df} \{G' \mid G \vdash^*_{(\mathsf{AH}+\mathsf{RAE})} G'\},$$

which are derivable from G by sequences of admissible assignment hoistings and redundant assignment eliminations with the relation "better" being defined as in Definition 4.4.1 (cf. [54]).

4.5 Optimality of MAP

As recalled above, the elementary transformations of PDCE or PRAE can be applied in any order without affecting the program finally resulting (cf. [53, 54]). 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 as illustrated in Figure 4.11 for PDCE. G and G_{opt} denote the original and the optimal program of the program universe $\mathcal{G}_{pdce}$, respectively. Since program versions are getting "better" from left to right, G_{opt} is located at the rightmost position. For PDCE and for PRAE we know that there exists such an unique, optimal program G_{opt} and that we can reach it independently of the transformation sequence. In Figure 4.11 both transformation sequences (AS DCE AS DCE) and (DCE AS DCE) terminate with G_{opt}. Different transformation sequences have no influence on the result but only on the number of iterations required to reach the optimum. Unfortunately, this property gets lost as soon as all four elementary transformations are interleaved as required for MAP and shown by the following Theorem.

Theorem 4.5.1 (No global optimum for MAP)
Let $\mathcal{G}_{mix} =_{df} \{G' \mid G \vdash^*_{(\mathsf{AS}+\mathsf{DCE}+\mathsf{AH}+\mathsf{RAE})} G'\}$ *denote the set of programs derivable from* G *by*

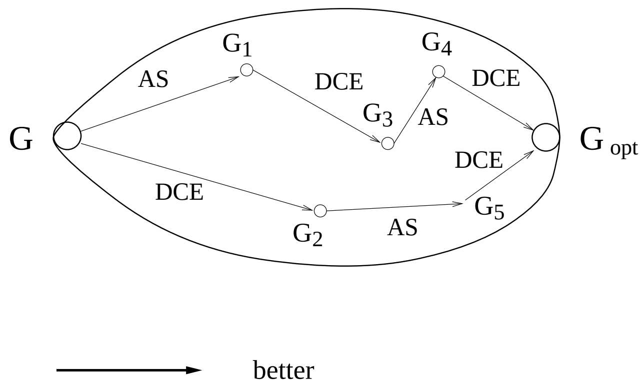


Figure 4.11: "Globally best" program for PDCE and PRAE.

means of admissible assignment sinkings (hoistings) or dead (redundant) assignment eliminations. Then $\mathcal{G}_{mix}$ does not necessarily have an unique, optimal program based on the relation "better" as defined by Definition 4.4.1.

Proof.

Consider the flow graphs of Figure 4.12. Applying PDCE first we arrive at the program of Figure 4.12(b), applying PRAE first we arrive at the program of Figure 4.12(c). While each path through the program fragment of Figure 4.12(c) contains exactly one mapping assignment, there is a path through the fragment of Figure 4.12(b) being free of mapping assignments, and another one containing two. Hence, both program versions are incomparable and none of them is optimal. ∎

As pointed out by the example of Figure 4.12, PDCE and PRAE influence each other by mutually removing opportunities for their respective counterpart. Note that both programs are invariant under further admissible assignment motions and dead (redundant) assignment eliminations. Figure 4.13 illustrates the new characteristic.

Again, program versions are getting "better" in the sense of Definition 4.4.1 from left to right. However, we do not have an unique, optimal program G_{opt}, but a set of local minima G_{loc1} to G_{loc4} which usually are not comparable. Each of the local minima are better on some paths, but worse on others. Under the assumption that the probability for each path to be executed is the same, a "best" program does not exist any more.

In fact, we conjecture that like in the example of Figure 4.12 "maximally transformed" programs are either identical (up to irrelevant reorderings in basic blocks) or incomparable with respect to the order of Section 4.4.3 counting assignments for every assignment pattern separately. But programs may be considered incomparable for which an order according to the intuitive notion of "better" exists as demonstrated in the example of Figure 4.14. The programs of Figure 4.14(b) and 4.14(c) result from program 4.14(a) by beginning with PRAE and PDCE, respectively. On paths starting from node **1** program 4.14(c) contains less remappings

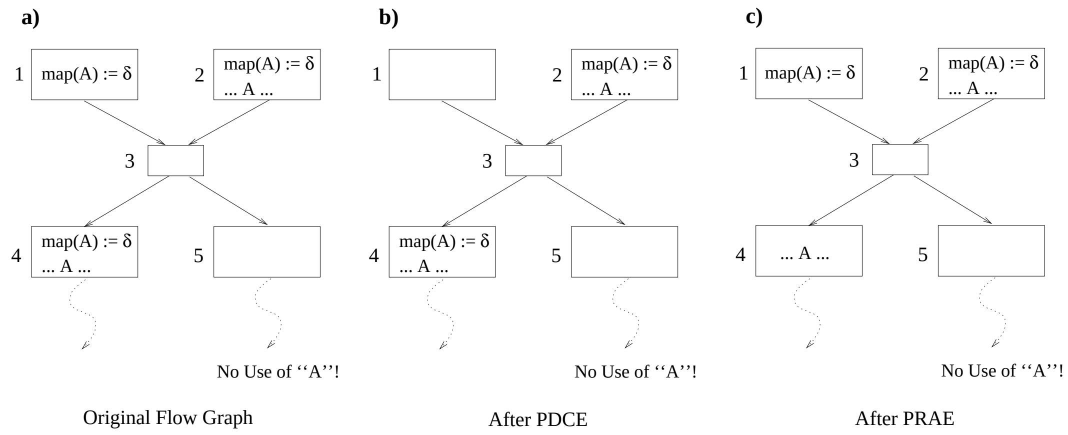


Figure 4.12: Interdependences between PDCE and PRAE.

than program 4.14(b). On path (3,4,5,7) the same remappings are required for both versions. On path (3,4,5,6), however, the mapping assignments map(A) := BLOCK(i) and map(B):= α_A^B are executed in 4.14(b), whereas in 4.14(c) the assignment map(A) := BLOCK(i) is performed twice. Hence, both programs are incomparable with respect to the relation "better" of Section 4.4.3. However, taking execution costs into account, both programs would perfectly be comparable, if the execution costs of map(A) := BLOCK(i) and map(B):= α_A^B coincide. Then, the program of 4.14(c) is strictly "better" than that of 4.14(b); a fact, which is not reflected by the order of Section 4.4.3. Unfortunately, validating the equality of costs of different mapping assignment patterns is in general not trivial and requires a performance prediction evaluating program paths statically. Thus, it is worth noting that in program 4.14(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 4.14(c) the second remapping is redundant. By means of *mapping assignment masking* (cf. Section 4.7.4) this unnecessary remapping can be suppressed at the price of a much cheaper runtime test implying that program 4.14(c) is "better" than program 4.14(b) without having to refer to the relative costs of map(A) := BLOCK(i) and map(B):= α_A^B. Note, however, that "better" relies now on a specific code generation strategy.

4.6 MAP Transformation Order

In spite of the difficulties with optimality, 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". Different transformation sequences, however, will usually come up with different locally best programs. In the following, we therefore concentrate on the question in which order to apply the elementary transformations to achieve fast stabilization. Obviously, AS- and AH-steps

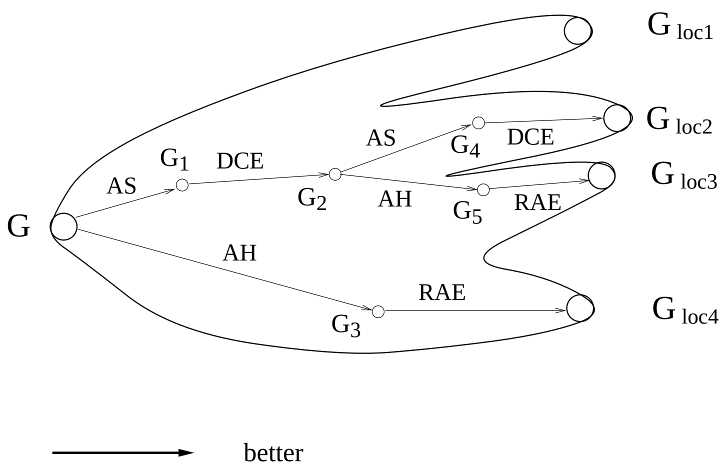


Figure 4.13: "Locally best" programs when applying both PDCE and PRAE.

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 B 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? The following facts suggest starting with PDCE. First, the situation displayed in the example of Figure 4.2, where PDCE is a prerequisite for enabling PRAE, can be considered quite typical for data-parallel programs. Second, there exists an important difference between PDCE and PRAE as stated in Lemma 4.6.1.

Lemma 4.6.1 (PDCE – PRAE **Interplay**)

1. PRAE *never creates (partially) dead assignments.*

2. PDCE *may create (partially) redundant assignments.*

Proof.

1. Consider the transfer function of (partial) dead code analysis (cf. Table 4.5): $\mathbf{LocMod}_n(v) + \mathbf{DDO}_n(v) * \overline{\mathbf{LocUsed}_n(v)}$. PRAE consists of RAE and AH. Obviously, RAE does not affect **LocUsed** and RAE cannot cause any $\mathbf{LocMod}_n(v)$ to change from *true* to *false*. Thus, RAE cannot create new (partially) dead assignments. The same holds for AH, since AH is blocked by any occurrence of the left-hand side of the assignment under consideration.

2. We show it with an example. Consider Figure 4.10. Sinking of the mapping assignments just after the subroutine call in node **3**, turns both mapping assignments just before the subroutine call to be partially redundant.

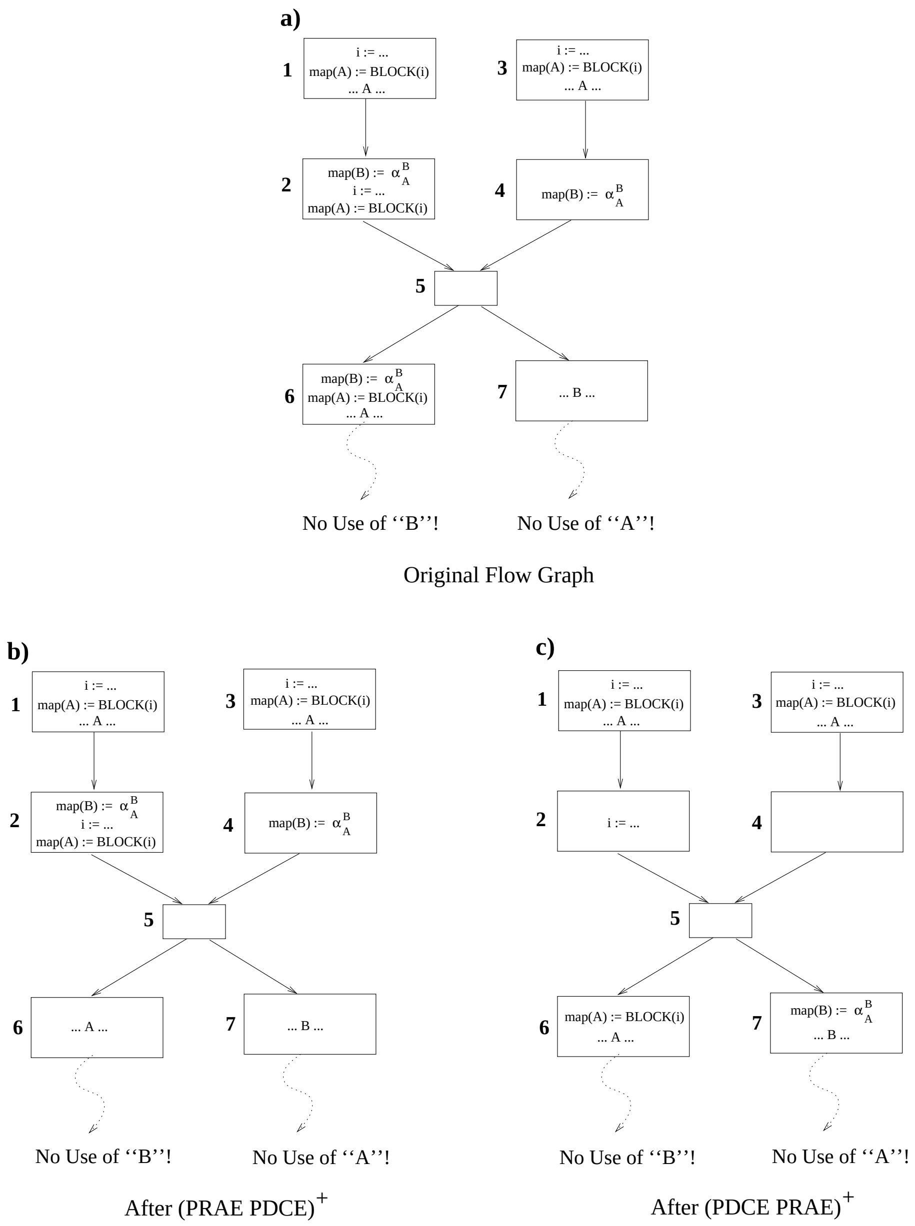


Figure 4.14: "Maximally" transformed programs are sometimes comparable.

■

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 4.2 where already PDCE followed by PRAE terminates with a locally best program.

4.7 MAP-Hierarchy

Before presenting the hierarchy, we remark that the computational complexity of PDCE and PRAE, and hence of MAP 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 mapping assignments this can be achieved by a simple *preprocess* which focuses on alignments and variables occurring in mapping specifications.

4.7.1 The Preprocess – Decoupling Mapping Assignment Patterns

Starting point of the preprocess are alignments and variables in mapping specifications. They are treated 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 mapping definition* analysis yields that array A is distributed by BLOCK.
- *Variables in mapping specifications*: Variables used in mapping specifications like variable i in BLOCK(i) or CYCLIC(i) may prevent mapping 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 the preprocess for decoupling assignment patterns is not a prerequisite of our approach, rather a means of improving efficiency. Moreover, there is a trade-off between the costs of the preprocess and the number of iterations subsequently saved. However, reaching mapping definition and constant propagation analysis, on which it relies, are usually also exploited for other optimizations, and are thus in part for free.

4.7.2 Pure MAP

Pure MAP focuses on the placement of mapping assignments. It is the iterated sequential composition of the algorithms for PDCE and PRAE applied to the mapping assignment patterns of $\mathcal{D}$:

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

The iteration process 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.

4.7.3 Full MAP

Sine the Pure MAP-algorithm focuses on mapping assignment patterns only, it does not capture second-order effects due to ordinary assignments. As a consequence the elimination of partially dead and partially redundant mapping assignments can be prevented by not considering the interdependences with ordinary assignments (cf. Figure 4.15(b) where the mapping assignment map(A) := CYCLIC(k) is blocked by the ordinary assignment k at node **4**). The Full MAP-algorithm 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 $\mathcal{P}$:

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

Figure 4.15 illustrates the different power of the algorithms of Pure MAP and Full MAP by means of the example of Figure 4.10. Whereas in this example a partially dead mapping assignment remains in the program after Pure MAP (cf. Figure 4.15(b)), the algorithm of Full MAP succeeds in eliminating all partially dead and partially redundant mapping assignments. Note, however, that also the transformationally weaker algorithm of Pure MAP succeeds here in removing all mapping assignments from the loop.

4.7.4 Customized MAP-Variants: A Hierarchy of MAP-Algorithms

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

Enhancing the efficiency

As pointed out, Pure MAP focuses on mapping patterns. In essence, this relies on the ratio that in practice mapping assignments are used in quite a restricted manner only. Thus, it is likely to get a reasonable amount of the effect of Full MAP much more efficiently. In addition to limiting the statement patterns considered, this suggests to also limit the number of iterations performed. The extreme variant here is the one-step heuristic focusing on mapping patterns:

$$\mathsf{MAP}^{one-step}_{pure} \equiv (\mathsf{AS}_{\mathcal{D}}\ \ \mathsf{DCE}_{\mathcal{D}})\ \ (\mathsf{AH}_{\mathcal{D}}\ \ \mathsf{RAE}_{\mathcal{D}})$$

We remark that $\mathsf{MAP}^{one-step}_{pure}$, which applies the elementary transformations only once, fails in capturing any second-order effect. 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 4.2 of Section 4.2 even the one-step heuristics yields the optimal result, and in Figure 4.10 it removes all mapping assignments from the loop.

Masking Mapping Assignments

All MAP-algorithms can be combined with *mapping assignment masking*. This means to prefix mapping assignments by masks which ensure that they are only performed if the actual

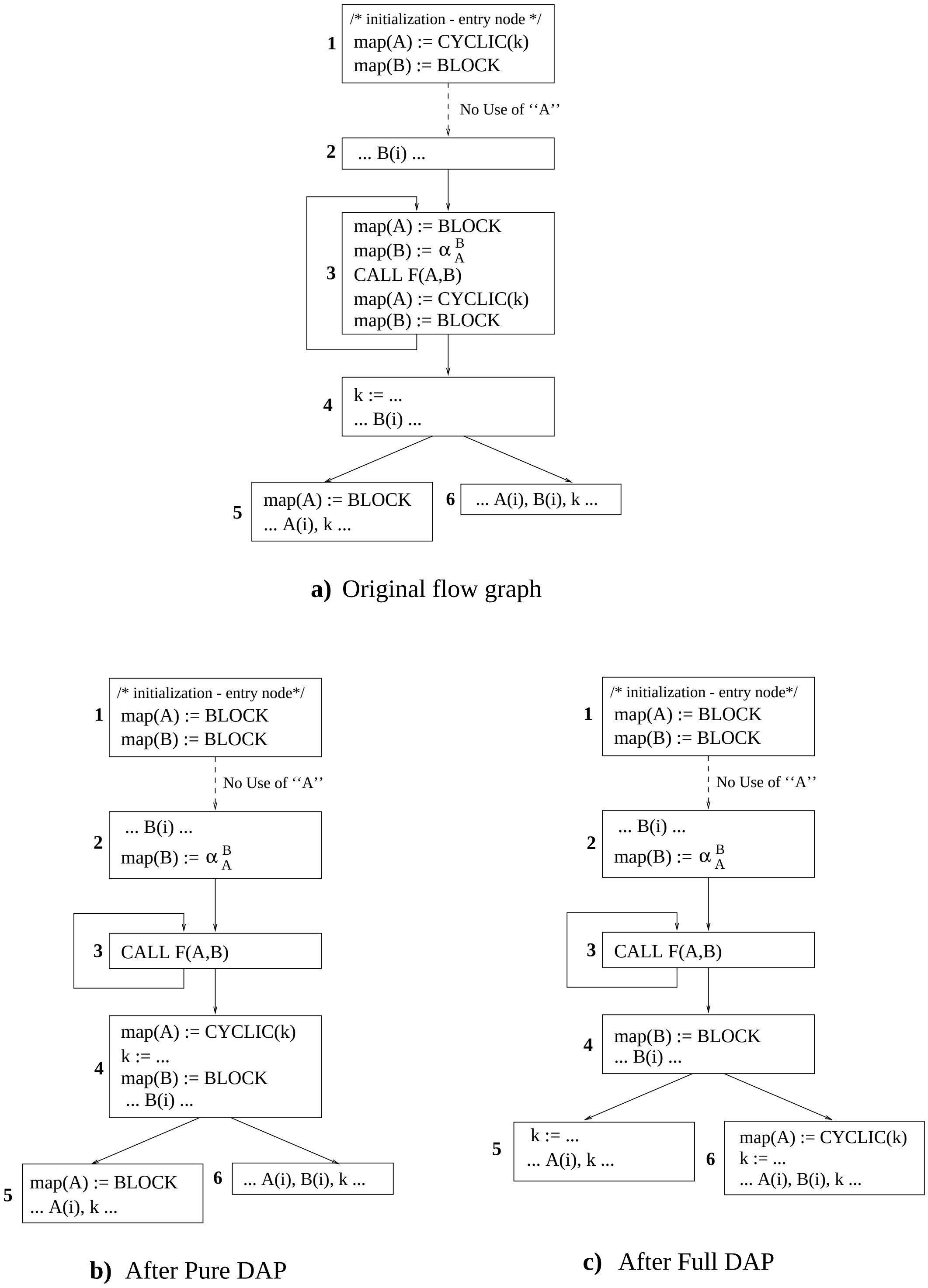


Figure 4.15: Pure MAP vs. Full MAP

mapping differs from the mapping which would be assigned to by the statement under consideration. Technically, $\mathsf{map(A)} := rhs$ is replaced by: $\mathsf{if}\ (\mathsf{map(A)} \neq rhs)\ \mathsf{map(A)} := rhs$. Evaluating a mask consists of determining rhs and comparing it with the actual mapping of array A. The effort required depends on the concrete implementation of the runtime system and may differ significantly for regular mappings like BLOCK and irregular mappings with mapping arrays, but usually it is much cheaper than the mapping statement it guards. Thus, masking can be used to reduce the overhead caused by partially redundant mapping assignments remaining in a program after MAP. We remark, however, that masking introduces overhead for program executions for which the masked occurrence is not redundant. Note that masking is not possible remaining partially dead assignments: they have to be executed in any case.

Enhancing the power

The transformational power of every MAP-algorithm 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 [53] 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 mapping patterns the notions of "dead" and "faint" coincide due to the syntactic limitations applying to their right-hand side terms.

4.7.5 Local Optimality Results

Let G_{pure} and G_{full} be the programs resulting from our algorithms MAP_{pure} and MAP_{full}, respectively. Denoting additionally the sets of programs derivable from G by applying the four elementary transformations of Pure MAP and Full MAP in any order by

$$\mathcal{G}_{mix_\mathcal{D}} =_{df} \{G' \mid G \vdash^*_{(\mathsf{AS}_\mathcal{D}+\mathsf{DCE}_\mathcal{D}+\mathsf{AH}_\mathcal{D}+\mathsf{RAE}_\mathcal{D})} G'\} \quad \text{and}$$

$$\mathcal{G}_{mix} =_{df} \{G' \mid G \vdash^*_{(\mathsf{AS}+\mathsf{DCE}+\mathsf{AH}+\mathsf{RAE})} G'\},$$

we have:

Theorem 4.7.1 (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.

Proof.

Based on the optimality considerations of Section 4.4.3 and 4.4.4 it is easy to see that G_{full} and G_{pure} are locally best, since the algorithms MAP_{pure} and MAP_{full} stop when the elimination steps fail to remove some code. ■

Actually, this is almost the best we can expect for G_{full} (G_{pure}) in $\mathcal{G}_{mix}$ ($\mathcal{G}_{mix_{\mathcal{D}}}$). As shown by Theorem 4.5.1, $\mathcal{G}_{mix}$ ($\mathcal{G}_{mix_{\mathcal{D}}}$) 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., $\mathsf{MAP}^{smpl} \equiv (\mathsf{PDCE}\ \mathsf{PRAE})$) 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

$$\mathcal{G}_{pure} =_{df} \{G' \mid G \vdash^*_{(\mathsf{AS}_{\mathcal{D}}+\mathsf{DCE}_{\mathcal{D}})} \bar{G} \vdash^*_{(\mathsf{AH}_{\mathcal{D}}+\mathsf{RAE}_{\mathcal{D}})} G'\} \quad \text{and}$$

$$\mathcal{G}_{full} =_{df} \{G' \mid G \vdash^*_{(\mathsf{AS}+\mathsf{DCE})} \bar{G} \vdash^*_{(\mathsf{AH}+\mathsf{RAE})} G'\},$$

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:

Theorem 4.7.2 (2nd Optimality Theorem)
G^{smpl}_{full} and G^{smpl}_{pure} are optimal in $\mathcal{G}_{full}$ and $\mathcal{G}_{pure}$, respectively.

Proof.

Let us proof optimality for G^{smpl}_{full} (for G^{smpl}_{pure} it can be done analogously) and let us consider an arbitrary program G. According to Section 4.4.3 and [53], MAP^{smpl} stops after PDCE with a program version G_{pdce} which is optimal in $\{\bar{G} \mid G \vdash^*_{(\mathsf{AS}+\mathsf{DCE})} \bar{G}\}$. Similarly, we have based on Section 4.4.4 and [54] that MAP^{smpl} stops after PRAE with a program version G_{prae} which is optimal in $\{G' \mid G_{pdce} \vdash^*_{(\mathsf{AH}+\mathsf{RAE})} G'\}$. Hence, putting both steps together, we have that $G_{prae} \equiv G^{smpl}_{full}$ and G^{smpl}_{full} is optimal in $\mathcal{G}_{full}$. ∎

In this chapter our main goal was to develop algorithms which terminate with "optimal" solutions. In order to reach our goal we gave up motion restrictions, applied transformations iteratively, and considered ordinary assignments as well. However, we treated flow graphs separately missing opportunities for interprocedural optimizations. In the following chapter we will extend the MAP-approach interprocedurally.

Chapter 5

Interprocedural MAP

Procedures are a major means to decompose programs into small, manageable units and to improve in this way readability and maintainability. Support of interprocedural analysis and transformations is necessary in order to encourage programmers in writing modular programs. Generally, there exist four different approaches how to proceed in the presence of a procedure call p.

1. Procedure p is not analyzed and worst-case assumptions are made at the call site. This is identical to the approach we have taken so far which is referred to as intraprocedural approach. It guarantees semantically correct results, but ignores interprocedural information completely.

2. Procedure p is analyzed in a *flow-insensitive* way in order to improve the worst-case assumptions made at the call site. The term *flow-insensitive* means that the control flow paths inside the procedure are not taken into account. Standard intraprocedural techniques can be used with the refined assumptions. Obviously, flow-insensitive analysis can be done efficiently and the results of our first approach improved in this way. Flow-insensitive analysis is discussed in [84] and [67].

3. An appropriately modified procedure body of p textually replaces the procedure call. This process is called *inline expansion* [35, 4]. In fact, inline expansion transforms interprocedural problems into intraprocedural ones which allows the application of standard intraprocedural techniques. Since a procedure may be called in different contexts and inline expansion places copies of them in more specific ones, this approach offers in general the best opportunities for optimizations. But problems arise in case of direct or indirect recursions. Another drawback is that the body of a procedure is no longer analyzed once, but each copy of the procedure is dealt with separately. In [73] it has been shown that the compilation has been slowed down often by a factor of 10. Inline expansion is an interesting approach beneficial in some situations, but cannot replace "real" interprocedural techniques in general.

4. The control flow of the called procedure p is taken into account resulting in interprocedural control flow paths. Analyses and transformations are defined on interprocedural

flow paths analogously to its intraprocedural counterparts. Intraprocedural frameworks like monotone data flow systems [38] have to be extended to interprocedural frameworks appropriately [56] in order to handle interprocedural flow paths resulting in "real" interprocedural techniques. In the following we will describe how to enhance MAP in this way interprocedurally.

Interprocedural MAP addresses the problem of hoisting assignment patterns out of subprograms or moving them into subprograms in order to eliminate partially redundant or partially dead assignments, respectively.[1] Surprisingly at the first sight, it turns out that a straightforward adaption of the intraprocedural approach fails. A modified transformation has to be applied, since the intraprocedural one may lead to severe anomalies. After discussing the essential differences and analogies of MAP in the intra- and interprocedural case, we show how to overcome the anomalies in order to arrive at a powerful and flexible algorithm for interprocedural MAP(IMAP).

5.1 Motivating Examples

Figure 5.1 demonstrates the benefits of hoisting mapping assignments out of procedures. The array dummy argument 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, i.e., two mapping 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 second remapping is partially dead as it is only used along program continuations satisfying the condition of the branch statement in the main program P. Interprocedural PDCE succeeds in moving this remapping from F1 to the main program and subsequently out of the loop, as well, and to its only use site inside the branch statement. As a side-effect, the sinking of this mapping assignment turned the first one to be partially redundant within the loop. Thus, as a second-order effect, interprocedural PRAE succeeds in hoisting this remapping from F1 to the caller and places it just before the loop. In this way, the number of executed mapping assignments is reduced considerably. Moreover, this example illustrates another important feature: The mapping which has to be restored after a subroutine call is usually runtime 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 as demonstrated in Figure 5.1 where *restore* has been replaced by $(block, *)$ after assignment sinking.

Figure 5.2 shows that sinking of mapping assignments into procedures may be beneficial as well and, simultaneously, illustrates some properties of interprocedural code motion. Subroutine F is called twice in the main program each time preceded by a mapping assignment to the globally visible array A. Both mapping assignments are partially dead, since they are alive with respect to the then-branch of the if-statement of subroutine F and dead otherwise. This partially deadness can be eliminated by sinking the mapping assignments into subroutine F and placing it in the then-branch before the loop as shown in Figure 5.2(b). However,

[1] Note that hoisting of assignments can be seen as partially inlining.

(a) Basic mapping assignment generation.

```
! main program          ! internal                ! internal
program P               subroutine F1             subroutine F2(X,N)
real A(N,N)             ··· ! no remap,           integer N
···                   no A                        real X(N,N)
! A ≡ (block,*)         map(A):=(*,block)         !HPF$ distribute X(*,block)
do iter=1,MAXITER       call F2(A,N)              do j=1,N
  call F1               map(A):=restore              do i=2,N-1
end do                  ··· ! no remap,                 X(i,j)=X(i,j)/2.0 +
···                   no A                       +            (X(i-1,j) +
if (cond) then          end                      +             X(i+1,j))/4.0
  ··· A(i) ···                                       end do
end if                                            end do
end                                               end
```

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

```
! main program          ! internal                ! internal
program P               subroutine F1             subroutine F2(X,N)
real A(N,N)             ··· ! no remap,           integer N
···                   no A                        real X(N,N)
! A ≡ (block,*)         call F2(A,N)              !HPF$ distribute X(*,block)
map(A):=(*,block)       ··· ! no remap,           do j=1,N
do iter=1,MAXITER     no A                           do i=2,N-1
  call F1               end                             X(i,j)=X(i,j)/2.0 +
end do                                           +            (X(i-1,j) +
···                                              +             X(i+1,j))/4.0
if (cond) then                                       end do
  map(A):=(block,*)                               end do
  ··· A(i) ···                                    end
end if
end
```

Figure 5.1: Illustrating the power of interprocedural mapping assignment placement.

(a) Basic mapping assignment generation.

```
! main program
program P
  integer k
  real A(N,N)
  ...
  map(A):=cyclic(k)
  call F
  ...
  map(A):=cyclic(k)
  call F
end
```

```
! internal
subroutine F
  ··· ! no remap, no A
  if (cond) then
    do ...
       ⋮ ...A(i)...
    end do
  else
    map(A):=cyclic(2*k)
    ⋮ ...A(i)...
  end if
end
```

(b) Result of interprocedural PDCE.

```
! main program
program P
  integer k
  real A(N,N)
  ...
  call F
  ...
  call F
end
```

```
! internal
subroutine F
  ··· ! no remap, no A
  if (cond) then
    map(A):=cyclic(k)
    do ...
       ⋮ ...A(i)...
    end do
  else
    k=...
    map(A):=cyclic(k)
    ⋮ ...A(i)...
  end if
end
```

Figure 5.2: Illustrating sinking of mapping assignments and the influence of calling contexts.

sinking of assignment pattern map(A):=cyclic(k) into subroutine F is only possible, if at all call sites a sinking candidate of the same pattern is available. Otherwise a new computation would be inserted on some interprocedural path resulting in a semantically incorrect program. On the other hand, all assignment patterns map(A):=cyclic(k) preceding subroutine calls of F have to be moved into F. E.g. assume that only the first occurrence of assignment pattern map(A):=cyclic(k) is moved into subroutine F and the second one remains in the main program. The resulting program is semantically correct, but on flow paths entering subroutine F via the second call in the main program and continuing with the then-branch in F contain now two mapping assignments compared to one in the original program resulting in performance degradation. In the following we will discuss optimality and correctness in more detail.

5.2 Basic Definitions and Terminology

Here we extend the definitions of Section 4.1 to meet interprocedural requirements. First of all we deal now with flow graph systems. A program Π is a system $\langle \pi_1, \pi_2, \ldots, \pi_k \rangle$ of (mutually recursive) program units (procedures), 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. Thus, the variables of the main program are global variables of the subprograms, and can be accessed by them.

According to Section 4.1, program units are represented by flow graphs. A program Π is represented as system $M =_{df} \langle T_1, \ldots, T_k \rangle$ of flow graphs with disjoint sets of nodes N_i and edges E_i. $N =_{df} \bigcup \{ N_i \mid i \in \{1, \ldots, k\}\}$ and $E =_{df} \bigcup \{ E_i \mid i \in \{1, \ldots, k\}\}$ denote the sets of all nodes and edges of M, respectively. Moreover, $E_{call} \subseteq E$ denotes the set of all edges representing a subprogram call.

5.3 Intra- vs. Interprocedural MAP: Optimality and Motion Strategy

5.3.1 Optimality of the Interprocedural Component Transformations PDCE and PRAE

As already discussed, intraprocedurally both PDCE and PRAE can be organized optimally: for every argument program Π there exists a "globally best" program Π_{opt}, which can effectively be constructed, and which is *better* than any other program in the set of programs $\mathcal{G}$ derivable from Π by means of sequences of admissible assignment sinkings and dead code eliminations, or assignment hoistings and partial redundancy eliminations, respectively. Unfortunately, we loose this property when we combine both transformations PDCE and PRAE as required for MAP.

Interprocedurally, let us first discuss analogously to the intraprocedural case whether the component transformations, i.e. interprocedural PDCE and interprocedural PRAE (for short:

IPDCE and IPRAE), of interprocedural MAP (for short: IMAP) can be solved optimally before we examine IMAP itself. Consider program Π of Figure 5.3 for IPRAE which are variations of examples in [48]. Figures 5.4 and 5.5 display program versions Π' and Π'' which are derived from program Π by hoisting mapping assignments interprocedurally. Both Π' and Π'' are

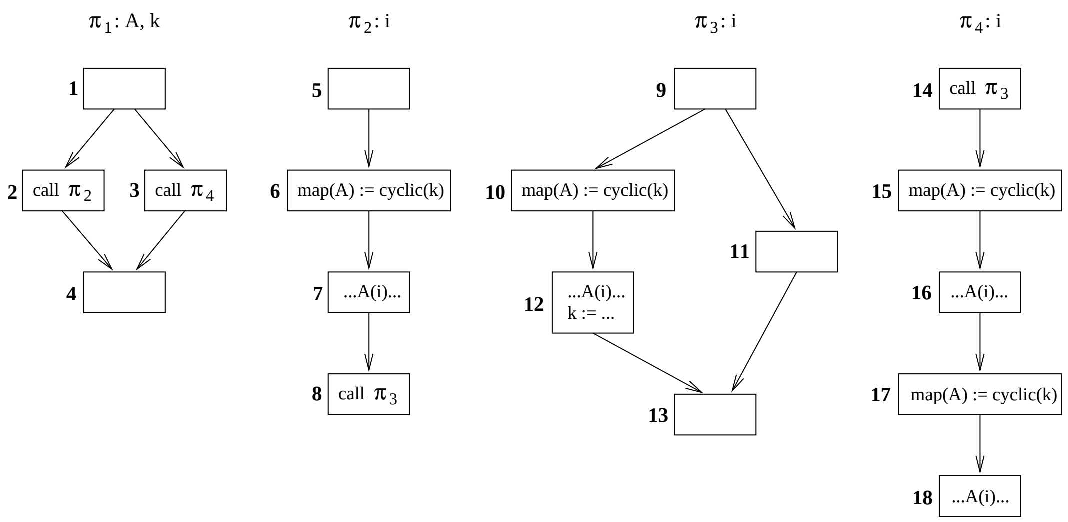


Figure 5.3: No optimality in general: The original program Π.

invariant under further mapping assignment hoistings and both are better than the original program Π. In Π flow paths via procedure π_2 contain either 2 or 1 mapping assignments and paths via π_4 either 3 or 2 mapping assignments depending whether the left or right branch of procedure π_3 is taken. For program Π' flow paths taking the right branch of procedure π_3 contain the same number of mapping assignments than Π. Taking the left branch of π_3, however, the number of mapping assignments can be reduced by 1, i.e. on flow paths via π_2 and left branch 1 mapping assignment is executed and on flow paths via π_4 and left branch 2 mapping assignments are performed. Program Π'' does not differ from Π on flow paths via procedure π_2. However, the number of mapping assignments on flow paths via π_4 can be reduced by 1 compared to Π, i.e. 2 and 1 mapping assignments are executed on the left and right branch, respectively.

Although both programs Π' and Π'' are better then Π, they themselves are incomparable. Whereas Π' contains 1 mapping assignment less than Π'' on the flow path via π_2 and left branch of π_3, Π'' executes 1 mapping assignment less than Π' on the flow path via π_4 and right branch of π_3. Hence, interprocedurally neither for PDCE nor for PRAE a globally best program exists in general. Generally, there are *incomparable minima* only as it is demonstrated by means of the program Π of Figure 5.3 for IPRAE.

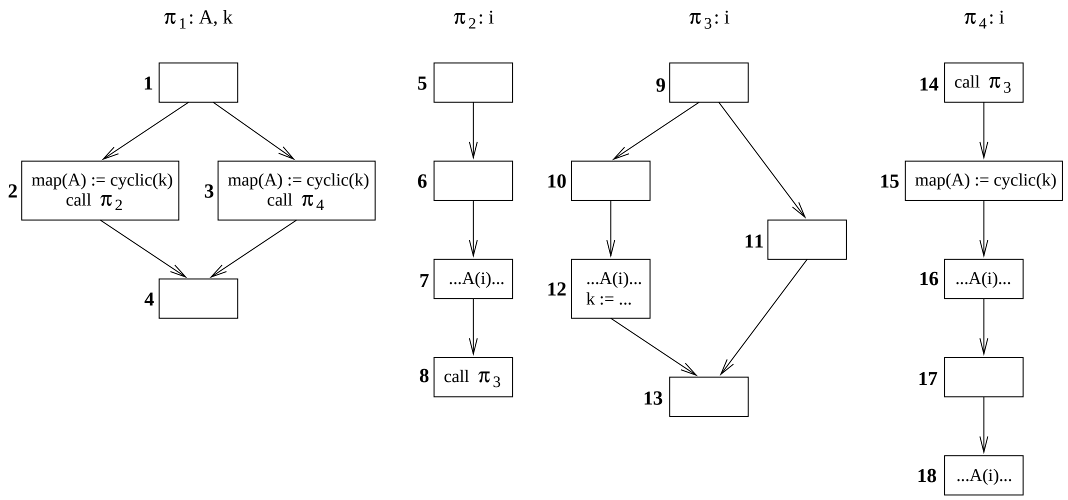


Figure 5.4: No optimality in general: A first minimal program Π'.

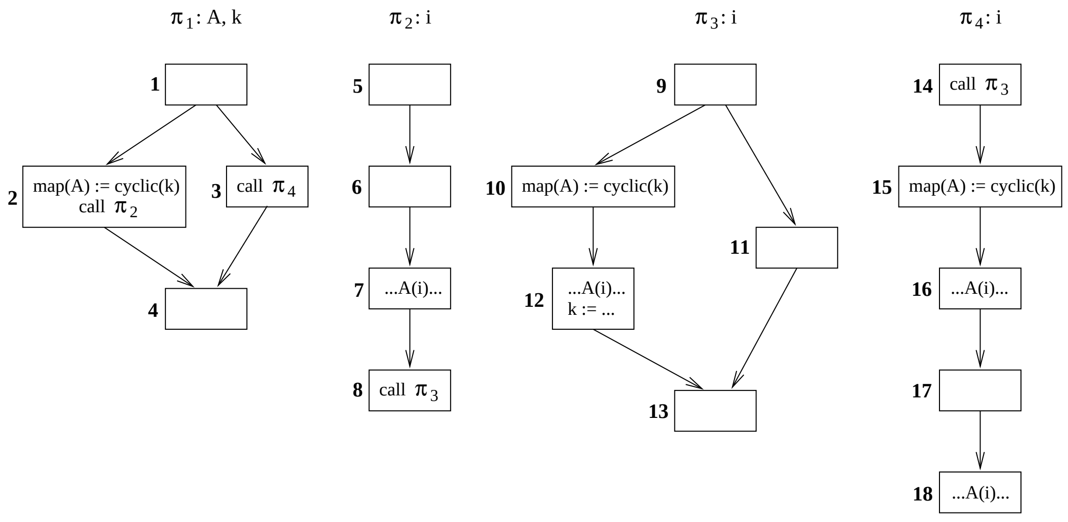


Figure 5.5: No optimality in general: A second minimal program Π''.

5.3.2 Adequateness of Moving Assignments "As-Far-As-Possible"

Although there does not exist in general an optimal program for the component transformations IPDCE and IPRAE, it is of course our aim to find motion strategies which result in "good" solutions. *Intraprocedurally*, the strategy underlying the hoisting and sinking of assignments is to move them *as-far-as-possible* because this maximizes the potential of redundant and dead code, respectively, without affecting the program semantics or performance (in terms of statements executed). In essence, for hoistings (sinkings) this means moving of assignments to program points having a predecessor (successor), where inserting an occurrence would affect the program semantics or performance. As recalled above, intraprocedurally this strategy leads to optimal results.

Interprocedurally, however, the very same strategy may lead to severe anomalies. This is illustrated by means of the program of Figure 5.6. Note that all mapping assignments are totally live. Thus, IPDCE does not have any effect. Note further that the mapping assignment in π_4 is *necessary*, i.e., it is not partially redundant. Using the as-far-as-possible hoisting strategy for the occurrences of map(A):=cyclic(k), results in the program depicted in Figure 5.7.[2] Since procedure π_2 calls π_3, map(A):=cyclic(k) is blocked in π_4 at the second call of π_3

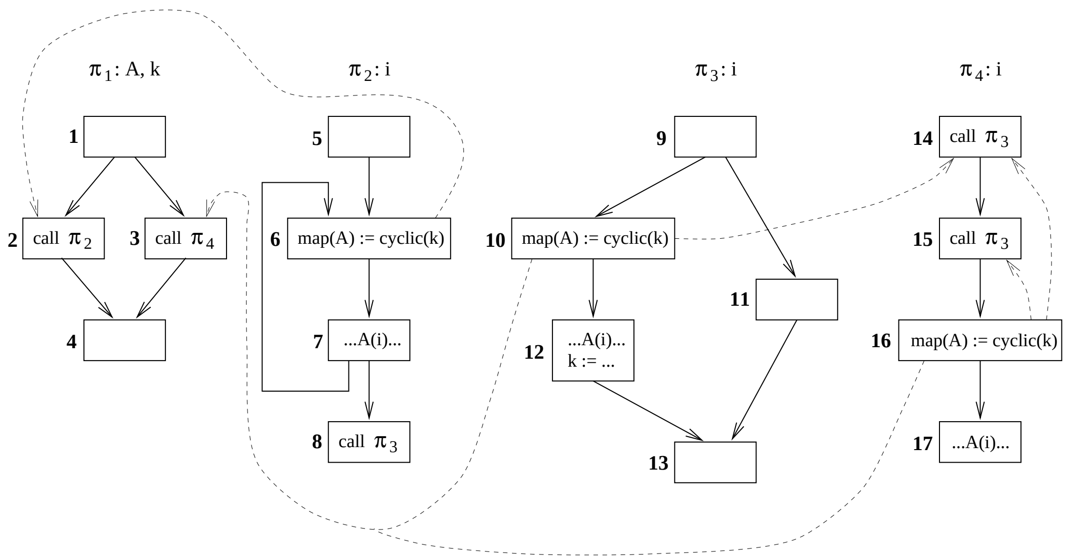


Figure 5.6: Moving assignments "as-far-as-possible" leads interprocedurally to anomalies.

because moving it inside π_3 would affect the performance of program executions calling π_3 via π_2. On the other hand, the occurrence of map(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 mapping assignment through π_3 and to place it just before the second call of π_3 in π_4. Exactly the same applies for the first call of π_3 in π_4 and, finally, map(A):=cyclic(k) is placed in π_1 and the hoisting process stops.

[2] In Figure 5.6, the hoisting is indicated by dashed lines.

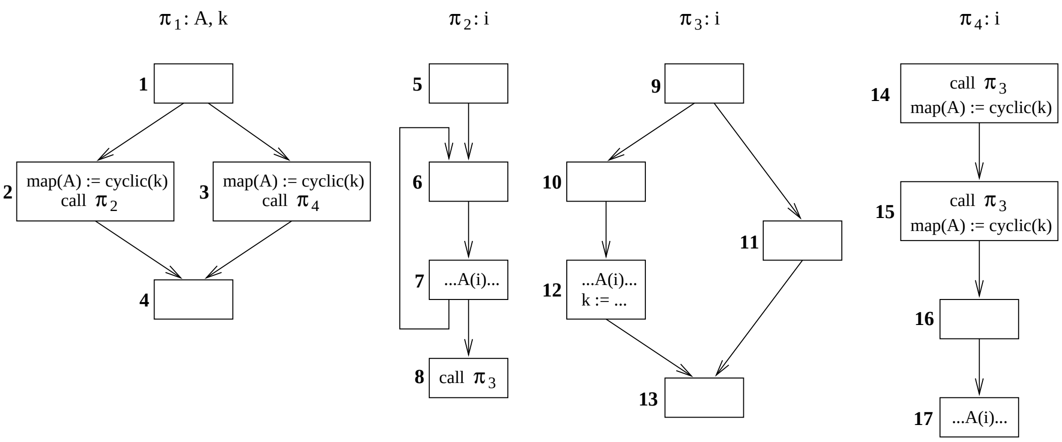


Figure 5.7: Having made the implicit transformation of Figure 5.6 explicit.

Unfortunately, however, this strategy increased the number of executed mapping assignments for subprogram π_4 (in fact, the number of mapping assignments can be increased arbitrarily by adding new subprogram calls of π_3 in π_4).

It is worth noting that the anomalies above only show up for "global" assignment patterns, i.e., for patterns, whose variables have all global scope. For assignment patterns involving local variables "mixed" situations as above, where movability of an assignment across a procedure call is justified in part by an assignment occurrence in the caller and an occurrence in the callee cannot happen as they refer to different variable incarnations. This point will be important for the construction of our IMAP-algorithm.

5.3.3 Uniform Handling of all Assignment Patterns

As already discussed in the previous chapter, ordinary assignments must be taken into account in general also in order to exploit the full power of MAP, since they can block the motion process of mapping assignments preventing thus their elimination. In case of ordinary assignments, *recursive* assignment patterns may occur. An assignment pattern is called *recursive*, if the left-hand side occurs in the right-hand side term like in u:=u+1. Note that recursive patterns are possible for ordinary assignments only. In HPF recursive mapping assignments are not possible. *Intraprocedurally*, all assignment patterns (ordinary and mapping assignments, recursive and non-recursive ones) have been handled uniformly. The admissibility of a particular sinking or hoisting strategy is independent of the shape of an assignment pattern.

Interprocedurally, however, recursive and non-recursive assignment patterns behave differently. This is illustrated in the example of Figure 5.8, which shows a situation being completely symmetrically for the assignment patterns u:=u+1 and map(A):=cyclic(k). An application of IPRAE to the flow graph system of Figure 5.8 only results in hoisting both assignments of node **17** across procedure call π_3 to node **15** as displayed in Figure 5.9. Note that the number of executed assignments has not been affected by this transformation, i.e. no performance gain

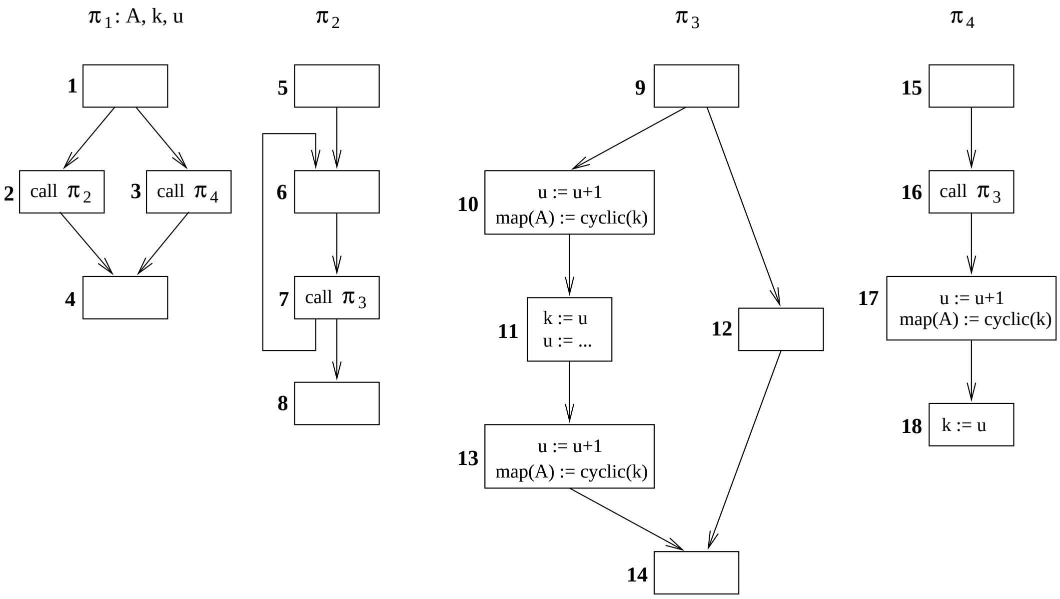


Figure 5.8: Illustrating the impact of "recursive" assignment patterns.

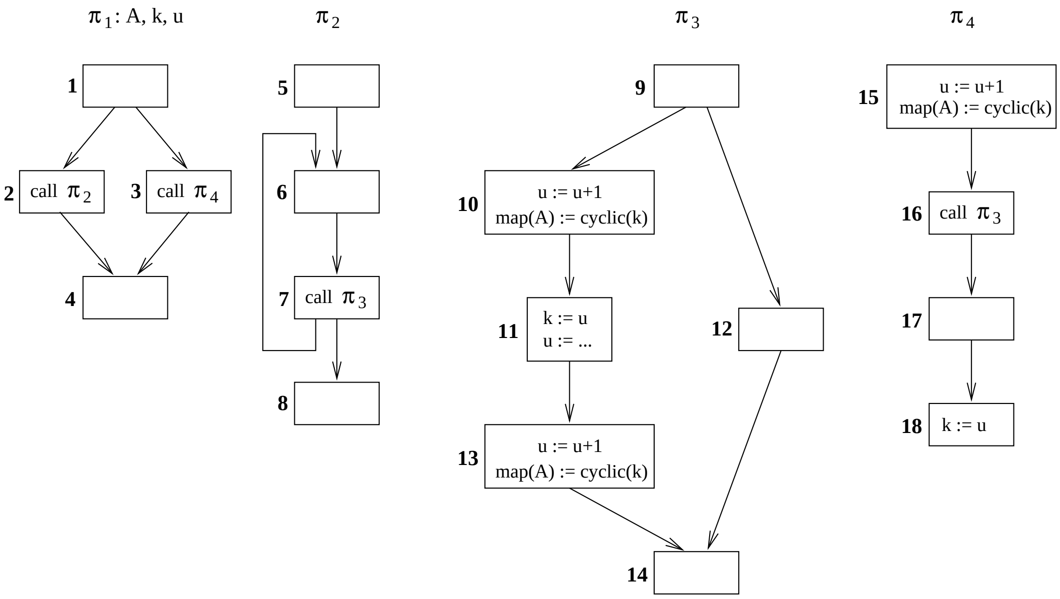


Figure 5.9: Treating "recursive" and "non-recursive" patterns analogously may lead to anomalies.

was achieved and the transformation unnecessary. Semantically, hoisting of the non-recursive assignment pattern map(A):=cyclic(k) is perfectly fine. However, the same transformation for the recursive assignment pattern u:=u+1 affects the program semantics. Consider program executions via π_4 and the left branch of π_3 and let us assume that u is initialized to 0 and reset to 0 again by the assignment in node **11**. Then in the original program of Figure 5.8, variable k gets assigned the value of 1 in node **11** and 2 in node **18**. In the transformed program of Figure 5.9, however, variable k is assigned the value 2 in node **11** and 1 in node **18** resulting in a semantically incorrect program. The transformation violated the semantics of the program, although the number of assignments have not been increased on any path. Obviously, a transformation involving a recursive assignment pattern will be always invalid, whenever on some path the number of that pattern has been increased as illustrated by Figure 5.6 where the number of assignments can be increased arbitrarily by adding new procedure calls.[3] Our goal is to arrive at an interprocedural motion strategy which allows a uniform handling of all assignment patterns similar to the intraprocedural approach.

5.4 Interprocedural Motion Strategy

In face of the missing optimality of IPDCE and IPRAE in general, which as a consequence also excludes global optimality of IMAP in general, and in face of the anomalies that may result from a naive adaption of the as-far-as-possible motion strategy for global assignment patterns, we focus in the remainder of this chapter on the development of a *practical* and *powerful* approach for IMAP, which in the absence of procedures reduces to its intraprocedural counterpart, avoids all the anomalies of a naive extension of the underlying intraprocedural approach, but simultaneously preserves the major benefits of its intraprocedural counterpart, in particular its (1) *flexibility*, (2) *extensibility*, and (3) *uniformity*, i.e., its capability of handling all assignment patterns in the same fashion.

Intuitively, this will be achieved by imposing an additional 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. As in the intraprocedural case, we thereby arrive at a hierarchy of IMAP-algorithms of varying power and efficiency allowing user customized solutions. Like their intraprocedural counterparts they are open for additional refinements for further enhancing their power and performance e.g. by *decoupling* of (mapping) patterns or *masking* of mapping assignments (cf. Section 4.7.4).

Fundamental is the introduction of the motion constraint $\mathcal{C}$ in Section 5.4.1 in order to avoid the sinking and hoisting anomalies discussed in Section 5.3. In Section 5.4.2 we stepwise develop the hierarchy of IMAP-algorithms.

[3] Note that there we discussed the anomaly of increasing the number of non-recursive mapping assignments resulting in performance degradation and did not address semantical problems arising thereof.

5.4.1 Avoiding Hoisting- and Sinking-Anomalies: The Motion Constraint $\mathcal{C}$

As illustrated in Section 5.3, 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 an additional constraint $\mathcal{C}$ on moving global assignment patterns across subprogram calls. Constraint $\mathcal{C}$ allows motion of global assignment patterns across procedure calls only, if no anomalies may occur which is ensured by two conditions.

Constraint 5.4.1 Motion Constraint $\mathcal{C}$

Motion of assignment patterns across procedure calls is allowed only, if one of the following two conditions hold:

1. *Uniformity Condition: The start node and end node of the callee satisfy the motion predicate under consideration.*
2. *Independence Condition: The callee and every procedure indirectly callable by it do not contain a statement blocking the assignment 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 5.6, where from the point of view of some call site (in Figure 5.6 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 5.6 the call site of π_3 in π_2) because the semantics or performance would be affected (in Figure 5.6 the performance of paths calling π_3 via π_2 following the left branch in π_3). The uniformity condition on start and end node is sufficient to detect these situations and to exclude the resulting anomalies. In Figure 5.6 e.g., it prevents moving the assignment pattern $\mathsf{map(A)} := \mathsf{cyclic(k)}$ across the subprogram call of π_3 in π_4. The uniformity condition would be satisfied, if the call of π_3 in procedure π_2 is followed by an assignment pattern $\mathsf{map(A)} := \mathsf{cyclic(k)}$. Figure 5.10 shows the modified program which fulfills the uniformity condition. Hence, the mapping assignments $\mathsf{map(A)} := \mathsf{cyclic(k)}$ are not blocked by the procedure call π_3 resulting in the transformed program displayed in Figure 5.11.

The second part of constraint $\mathcal{C}$ is important for avoiding that the motion constraint additionally imposed is unnecessarily restrictive. If the second part is satisfied, then there is no interference with other call sites of the callee under consideration. Hence, movability depends (as for an assignment pattern involving local variables) only on the context of the caller itself. This gives rise to consider it an *independence* condition. In Figure 5.12 we show a variation of Figure 5.6 which fulfills the independence condition. Although there is no hoisting candidate available just after the procedure call of π_3 in π_2, the assignment patterns $\mathsf{map(A)} := \mathsf{cyclic(k)}$ are not blocked at the procedure calls of π_3 in π_4, since the body of procedure π_3 does not block this pattern and thus is transparent for that particular pattern. The transformed program is displayed in Figure 5.13.

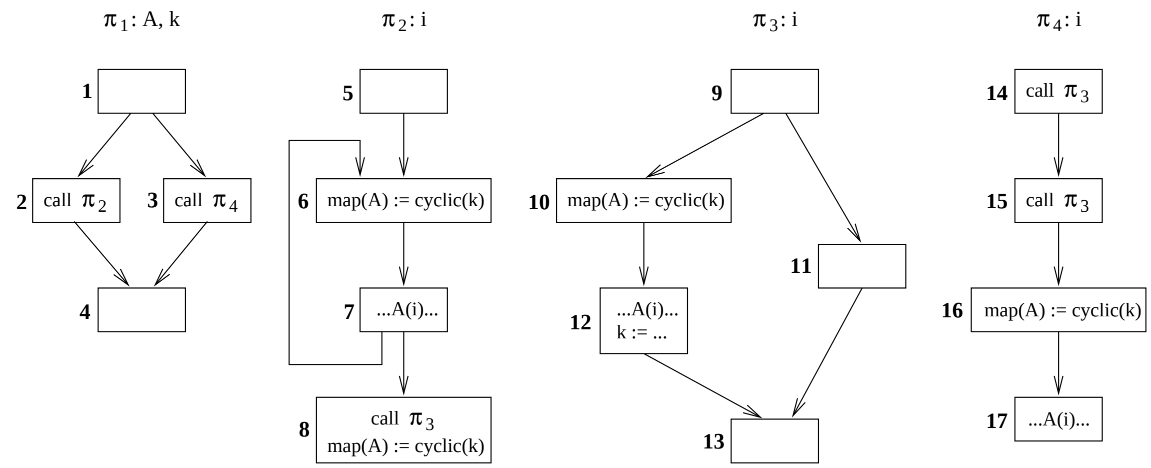


Figure 5.10: Uniformity condition fulfilled (variation of Figure 5.6).

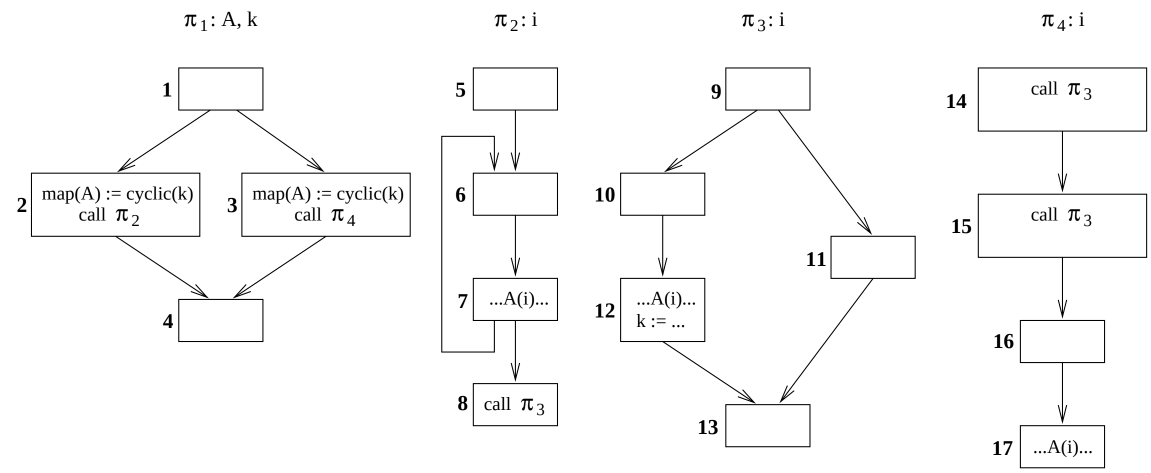


Figure 5.11: Having made the transformation of Figure 5.10 explicit.

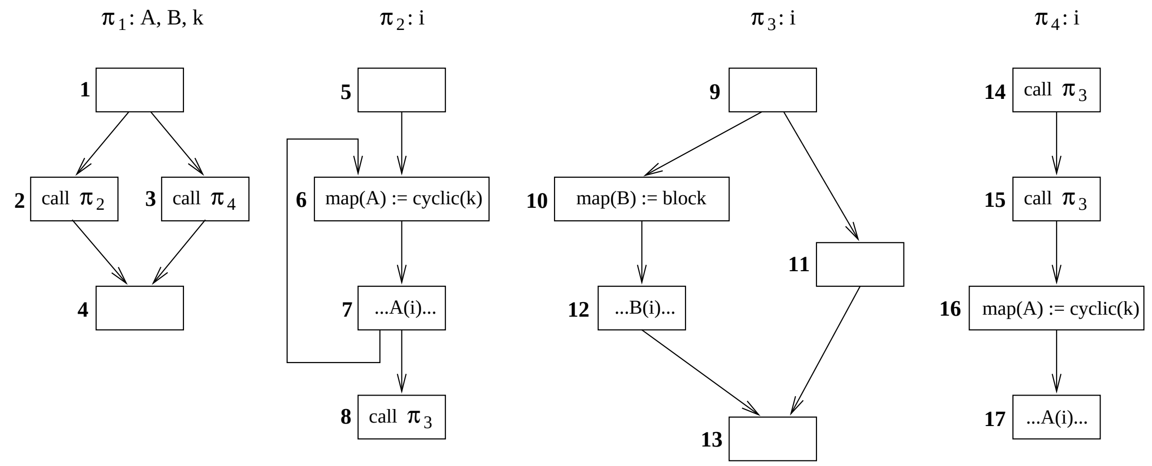


Figure 5.12: Independence condition fulfilled (variation of Figure 5.6).

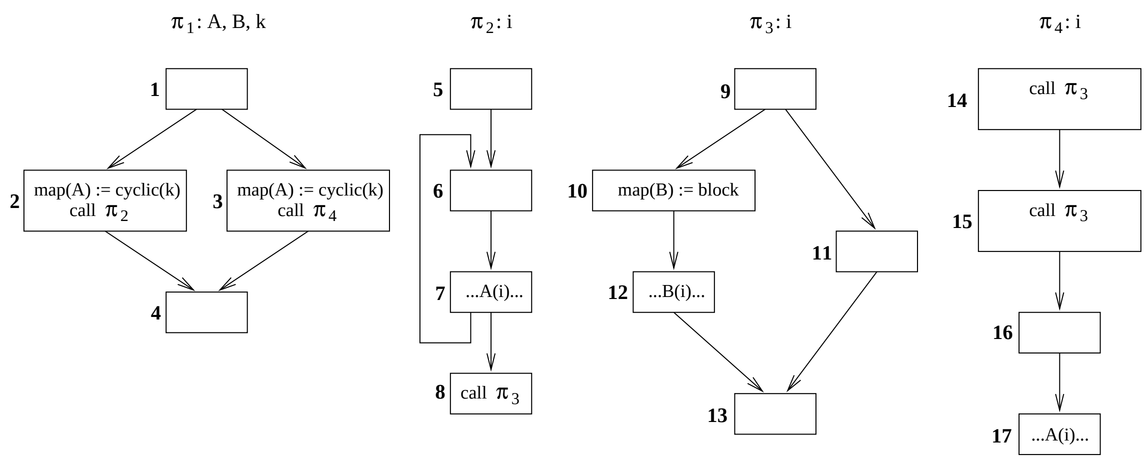


Figure 5.13: Having made the transformation of Figure 5.12 explicit.

5.4.2 A Hierarchy of IMAP-Algorithms

After introducing constraint $\mathcal{C}$, we can now present the *high-level* specifications of our interprocedural MAP-algorithms, which match the pattern of their intraprocedural counterparts. Using the regular-expression like notation of Chapter 4.4 our algorithms of *Pure* and *Full* IMAP constituting the kernel of our hierarchy of IMAP-algorithms which differ in the kinds of assignments patterns considered, are as follows.

Basic IMAP-Algorithms: Pure IMAP and Full IMAP

- **Pure IMAP**: As its intraprocedural counterpart, *Pure* IMAP focuses on mapping assignments, which are of primary interest for MAP. Below, this is indicated by the subscript $\mathcal{D}$. The algorithm for pure IMAP is then given by repeated applications of $\mathsf{IPDCE}^{\mathcal{C}}_{\mathcal{D}}$ and $\mathsf{IPRAE}^{\mathcal{C}}_{\mathcal{D}}$ to mapping assignment patterns until the program stabilizes:

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

  The superscript $\mathcal{C}$ here reminds to the additional constraint (cf. Section 5.4.1) imposed on the hoisting and sinking of global assignment patterns across subprogram calls. Hence, IPDCE and IPRAE expand to

  $$\mathsf{IPDCE}^{\mathcal{C}} \equiv (\mathsf{IAS}^{\mathcal{C}} + \mathsf{IDCE})^{+},\quad \mathsf{IPRAE}^{\mathcal{C}} \equiv (\mathsf{IAH}^{\mathcal{C}} + \mathsf{IRAE})^{+}.$$

- **Full IMAP**: The point of *Full* IMAP is to take all assignment patterns into account. As pointed out in Chapter 4.4 this is necessary in order to take care of interdependences between mapping assignments and ordinary assignments. The algorithm for full IMAP is thus given by:

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

In general, IMAP_{full} is more powerful than IMAP_{pure}, as it resolves second-order effects between all assignment patterns instead of those between mapping patterns only. This greater transformational power comes at the price of a greater computational complexity. This is the starting point for further refinements of the basic algorithms, which as in the intraprocedural case (cf. Section 4.7.4) results in a hierarchy of IMAP-algorithms of different power and efficiency allowing user customized solutions. On the one hand, the power can be enhanced by replacing partial dead code elimination by partial faint code elimination. On the other hand, the efficiency can be enhanced by decoupling assignment patterns and limiting the number of transformations performed.

5.5 Realization of the Interprocedural DFAs and the Transformations

In our discussion we strictly distinguished between transformation idea and DFA-specification and solution algorithm. DFA-specification and solution algorithm are a realization of the transformation idea. Iterative frameworks for solving intraprocedural data flow equations have already been extensively studied (cf. [38, 2]) and, hence, we did not touch this topic here (except some remarks on solving bidirectional data flow systems). Interprocedural data flow analysis, however, is more complicated and topic of current research.

In [55] an interprocedural DFA-framework has been presented which can be seen as an interprocedural extension of intraprocedural frameworks being based on the iterative approach. Analogously to the intraprocedural coincidence theorem [41, 42], an interprocedural coincidence theorem has been defined [56]. The interprocedural DFA-framework can be used in the very same way as its intraprocedural counterparts. The specifications for interprocedural DFAs can directly be fed into the generic computation procedure of the interprocedural DFA-framework of [55] underlying our approach. A realization of IMAP based on that DFA-framework has been presented in [49]. The constraint can be encapsulated into the generic algorithm for interprocedural DFA, and thus, it does not need explicitly taken care of when specifying the interprocedural counterparts of the hoistability and sinkability analyses.

The transformations induced by the solutions of the interprocedural DFA-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.

Chapter 6

Effectiveness of MAP/IMAP and Practical Experiences

The discussion of the benefit of MAP/IMAP ignores the effect of reducing ordinary assignments and concentrates on mapping assignments only whose runtime costs are significantly higher and can be very expensive. The main reason of considering ordinary assignments was motivated by dependences which could prevent reduction of mapping assignments. Let us consider assignment patterns $\alpha \in \mathcal{D}$ and paths $p \in [\mathbf{s}, \mathbf{e}]$. The execution time costs of mapping assignments on path p can be estimated easily by:

$$MapCost\,(p) = \sum_{\alpha \in p} cost(\alpha) * freq_p(\alpha)$$

with $cost(\alpha)$ denoting the execution time of mapping assignment α and $freq_p(\alpha)$ denoting the number of occurrences of α on path p. Our optimization aims at reducing the frequency $freq_p(\alpha)$. Let $freq'_p(\alpha)$ denote the mapping assignment frequency after running MAP/IMAP. Since we never increase the number of assignment patterns executed on a path or introduce new assignment patterns on a path, the following holds:

$$\forall p\ \forall \alpha.\ \ freq'_p(\alpha) \leq freq_p(\alpha)$$

The benefit of our optimization in terms of execution time is given by the difference between the unoptimized and the optimized program version for path p:

$$Benefit\,(p) = \sum_{\alpha \in p} cost(\alpha) * (freq_p(\alpha) - freq'_p(\alpha))$$

Besides specifying the benefit of MAP/IMAP in terms of execution time, relative performance numbers like the ratio of the execution time of the unoptimized version and the optimized version are important as well. Let $ExecTime\,(p)$ denote the execution time of path p which consists of the time required to perform mapping assignments ($MapCost\,(p)$) and the time required to execute the remaining statements exclusively mapping assignments ($RemCost\,(p)$):

$$ExecTime\,(p) = MapCost\,(p) + RemCost\,(p)$$

The performance improvement which can be achieved by our optimization on a path p is determined by the ratio of the unoptimized (*ExecTime*) and optimized execution time (*ExecTime'*):

$$Improvement(p) = \frac{ExecTime(p)}{ExecTime'(p)} = \frac{MapCost(p) + RemCost(p)}{MapCost'(p) + RemCost(p)}$$

where $MapCost'(p)$ denotes the execution time of mapping assignments on path p based on the reduced frequency $freq'_p$. Obviously, $Improvement(p) \geq 1$ since $MapCost(p) \geq MapCost'(p)$ for each path $p \in [\mathbf{s}, \mathbf{e}]$. $Improvement(p)$ will be close to 1 if $RemCost(p) >> MapCost(p)$. On the other hand if $RemCost(p) << MapCost(p)$, $Improvement(p)$ is determined by the ratio of $MapCost(p)$ and $MapCost'(p)$. Hence, we have the following situation:

$$1 \leq Improvement(p) \leq \frac{MapCost(p)}{MapCost'(p)}$$

Thus the performance improvement achieved by MAP/IMAP on a path p is heavily influenced by the amount of execution time spent for mapping assignments compared to the remaining execution costs on path p. Below we present some measurements which give an impression of the fraction of time required to perform computations and the time required to perform mapping assignments for some realistic computational kernels. Figure 6.1 shows the original and optimized version of an HPF code consisting of a computational kernel accessing a prescriptively mapped 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. The prescriptively mapped array results in insertion of mapping assignments at the call site as shown in Figure 6.1(a). MAP succeeds in moving both mapping assignments out of the loop. The resulting program is displayed in Figure 6.1(b). To show the benefit of our optimization, we measured for the original and the optimized program version the execution time of the do-loop of the main program. Let r_k with k denoting the number of iterations of the do-loop specify the number of remappings which are performed in the optimized version less than in the original one. Then r_k is given as follows: $r_{10} = 18$, $r_{55} = 108$, and $r_{100} = 198$. The measurement results are shown in Figure 6.2. 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 6.2 clearly illustrate the benefit of this kind of optimization for varying array sizes and loop iterations. The performance improvements vary from 154% for 10 iterations and $N = 1024$ to 331% for 100 iterations and $N = 256$.

As another example we ran on an Intel iPSC/860 hypercube and a Meiko CS-2 the unoptimized and optimized program versions of an Alternating-Direction-Implicit (ADI) integration kernel which is frequently used for solving partial differential equations. An ADI integration kernel consists of forward and backward sweeps along the rows followed by forward and backward sweeps along the columns. Hence, a row-wise data distribution requires communication for the sweeps along the columns but none along the rows – a column-wise data distribution requires communication vice versa. However, by choosing a row-wise distribution for the sweeps along the rows and a column-wise distribution for the column sweeps and redistributing the

(a) Basic mapping assignment generation.

```
! main program
program P
real A(N,N)
...
! start measurement
do iter=1,MAXITER
  map(A) := (*,block)
  call F1(A,N)
  map(A) := (block*)
end do
! stop measurement
...
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 MAP.

```
! main program
program P
real A(N,N)
...
! start measurement
map(A) := (*,block)
do iter=1,MAXITER
  call F1(A,N)
end do
map(A) := (block,*)
! stop measurement
...
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 6.1: A basic and an optimized program version of an HPF code fragment.

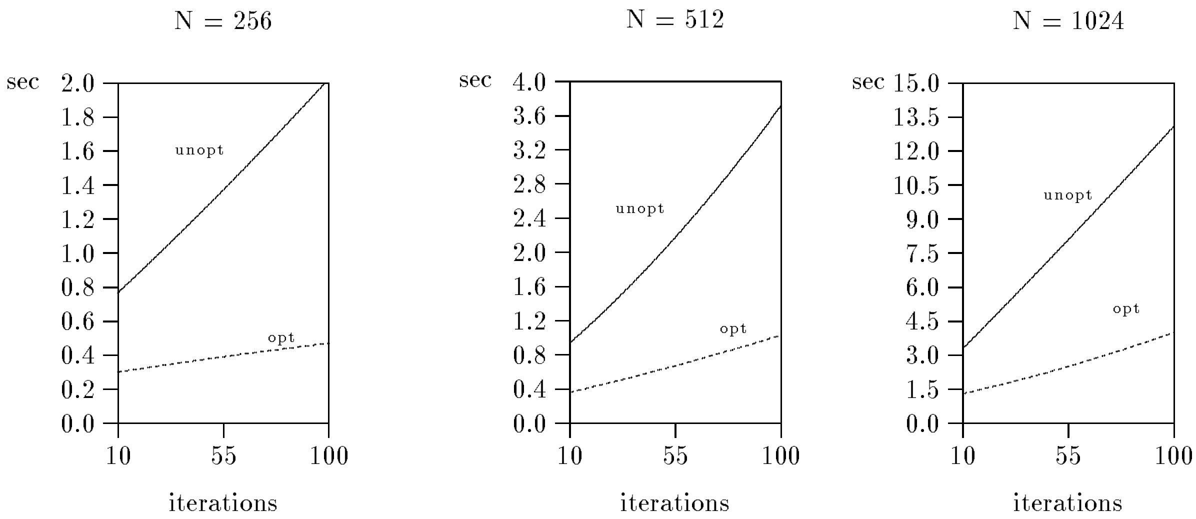


Figure 6.2: Measured run-times of Example of Fig. 6.1 on a Meiko CS-2 with 16 processors.

arrays between the two phases, no communication and synchronization is required within the sweeps at the expense of the remapping overhead.

In Figure 6.3 the unoptimized and optimized program version of ADI is shown in HPF with the required remappings. The ADI code chosen here has been used by U. Kremer [60] to analyze the runtime behavior of such codes with different data distributions on an Intel iPSC/860 (for details see [60]). Contrary to [60], we use subroutines to perform the sweeps along the rows and columns, respectively. Choosing subroutines instead of redistribute directives to perform remappings reflects also the current status of HPF 2.0, where dynamically distributed arrays have been moved from the base HPF language features to the Approved Extensions since an efficient implementation of dynamic data mappings can be difficult. As a consequence, most commercial compilers do not support dynamically mapped arrays at the present at all or in a rather inefficient way.

As shown in Figure 6.3, in the unoptimized version of the ADI main loop the arrays x, a, and b are remapped four times just before and just after the subroutine calls *rows* and *columns* in order to establish the mappings requested by the dummy arguments and to restore the original ones, respectively. First, PDCE is applied which recognizes the remapping at l_2 as dead. The remapping at l_4 is identified as dead within the loop but live with respect to the code following the loop. This partially deadness is eliminated by moving it just after the loop. The subsequently performed PRAE fails to improve the code further since no partial redundancies are present. Hence, the 12 remappings required within one iteration for the arrays x, a, and b in the unoptimized version are reduced to 6 remappings within the loop and 3 remappings after the loop in the optimized program version.

Figure 6.4 presents the execution times of the ADI integration kernel for various problem sizes N measured on an Intel iPSC/860 hypercube and a Meiko CS-2. For both machines 16 processors and the NX message passing library have been used. The number of remappings which are performed in the optimized ADI version less than in the unoptimized version is given by r_k with k denoting the executed iterations as follows: $r_1 = 3$, $r_2 = 9$, $r_5 = 27$, $r_{10} = 57$, $r_{50} = 297$, and $r_{100} = 597$. The optimization results in a performance improvement which varies on the hypercube from 16% for 1 iteration to 48% for 5 iterations. On the Meiko we achieved a performance improvement from 24% for 10 iterations to 38% for 100 iterations.

The dramatic performance improvement of the examples above is not surprising in the light of average remapping times. Ramaswamy et al. [72] 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.

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

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

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

(b) Subroutines performing the sweeps.

```
 SUBROUTINE rows(x,a,b)
 REAL, DIMENSION (:,:) :: x,a,b
 !HPF$ DISTRIBUTE (BLOCK,*) :: x,a,b
 ! sweeps along rows
   DO j=2,N
     DO i=1,N
       x(i,j)=x(i,j)-x(i,j-1)*a(i,j)/
+              b(i,j-1)
       b(i,j)=b(i,j)-  a(i,j)*a(i,j)/
+              b(i,j-1)
     END DO
   END DO
   DO i=1,N
     x(i,N)=x(i,N)/b(i,N)
   END DO
   DO j=N-1,1,-1
     DO i=1,N
       x(i,j)=(x(i,j)-a(i,j+1)*
+              x(i,j+1))/b(i,j)
     END DO
   END DO
END
```

```
 SUBROUTINE columns(x,a,b)
 REAL, DIMENSION (:,:) :: x,a,b
 !HPF$ DISTRIBUTE (*,BLOCK) :: x,a,b
 ! sweeps along columns
   DO j=1,N
     DO i=2,N
       x(i,j)=x(i,j)-x(i-1,j)*a(i,j)/
+              b(i-1,j)
       b(i,j)=b(i,j)-  a(i,j)*a(i,j)/
+              b(i-1,j)
     END DO
   END DO
   DO j=1,N
     x(N,j)=x(N,j)/b(N,j)
   END DO
   DO j=1,N
     DO i=N-1,1,-1
       x(i,j)=(x(i,j)-a(i+1,j)*
+              x(i+1,j))/b(i,j)
     END DO
   END DO
END
```

Figure 6.3: ADI integration kernel.

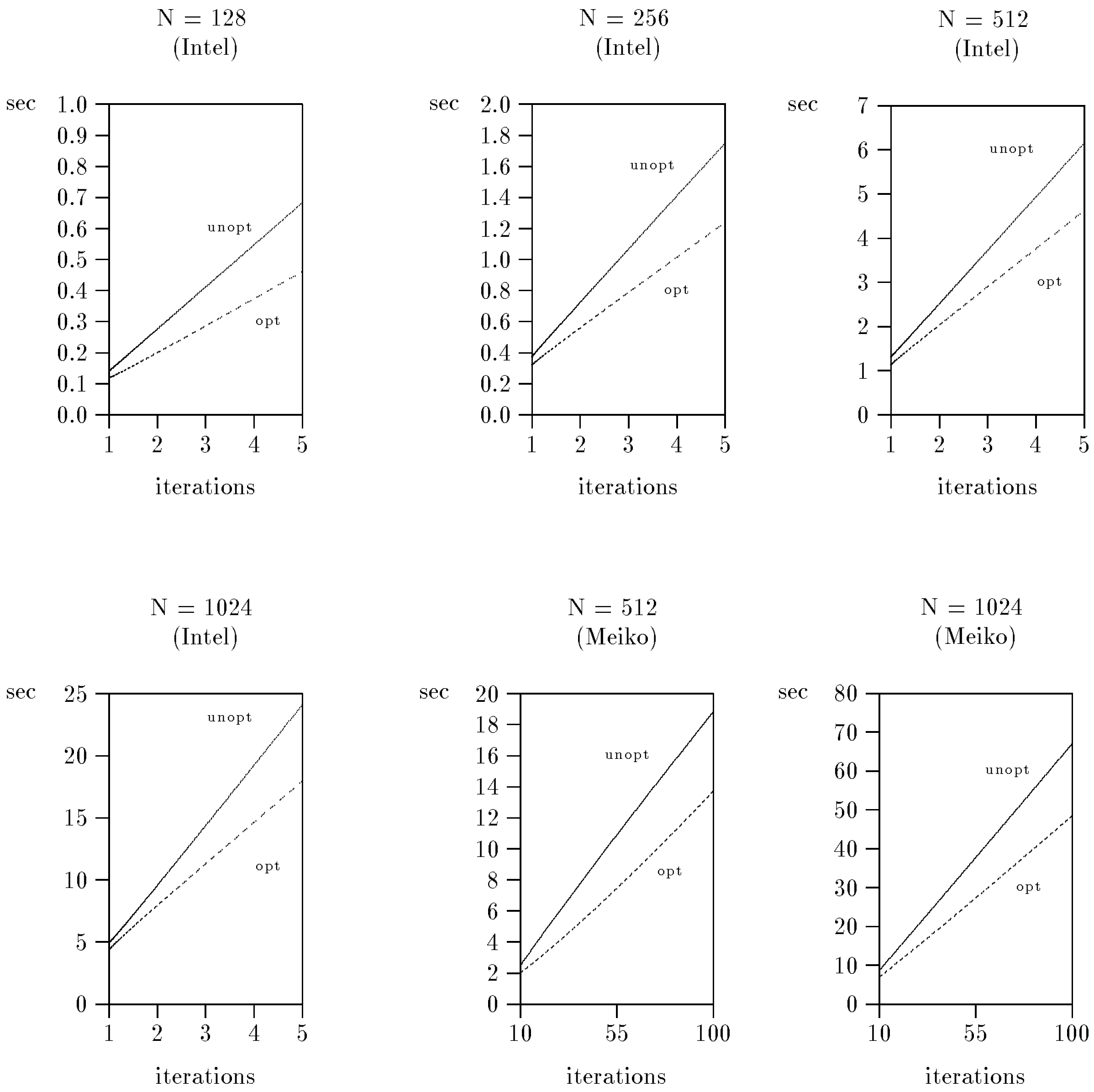


Figure 6.4: Measurements of ADI integration kernel for various problem sizes on different platforms.

Chapter 7

Related Work

7.1 Language Support for Data Remappings

Since data mapping may have a crucial impact on runtime performance and since many scientific applications consist of different computational phases each owning a best mapping, dynamic remappings have proven useful. Hence, data remappings have been an important issue in research in the field of language development and compiler support from the very beginning.

Blaze, Kali, KF1

Kali or more precisely Kali Fortran 1 (KF1) [63, 59] (with its predecessor BLAZE [62]) was one of the first languages which introduced data mapping annotations. The mappings supported include block, cyclic, and block_cyclic; alignments are not supported. The processor array specifies the range of processors; the actual processor number is chosen dynamically at runtime. KF1 supports dynamic remappings in a very general way by *distribute*-statements. But the language specification of KF1 is not complete, in particular with respect to procedure interfaces.

Hypertasking Directives

Hypertasking [5] is a domain decomposition tool developed at Intel that operates on C source code augmented by directives. In [6] extensions to Hypertasking are described which support dynamically resizeable and redistributable arrays; e.g. a *redistribute*-directive has been added to support remappings of arrays. Hypertasking was one of the first tools which supported dynamic remappings.

Fortran D

Fortran D [30] proposes a two level data mapping of the data space onto the local memories of the processing nodes. First the arrays are aligned to a virtual array, known as a decomposition, and then the decomposition is mapped to the processing nodes. The first mapping is referred to as *problem mapping* and the second one as *machine mapping*. Dynamic remappings are supported by *dynamic realignment* and *dynamic redistribution* statements: depending on their location align- and distribute-statements may be interpreted as executable statements rather than declarations.

Vienna Fortran

Vienna Fortran is a machine-independent language extension to Fortran 77 [83]. Its successor Vienna Fortran 90 is based on Fortran 90 [10]. Vienna Fortran provides the basic intrinsic distribution functions *block*, *cyclic*, *block-cyclic*, and *indirect*. New distribution functions may be defined by the user. Alignments are supported. Executable directives support dynamic redistributions and realignments, as well. At procedure boundaries a local view of data mappings is provided. A mapping may be inherited or an implicit remapping may take place depending on the declaration of the dummy argument. The Vienna Fortran Compilation System (VFCS) [9] translates Vienna Fortran source programs into message passing code.

High Performance Fortran: HPF, HPF 2.0

High Performance Fortran (HPF) is the current de-facto standard. Many language proposals including above all Fortran D and Vienna Fortran contributed essentially to the definition of HPF. The first version of the language definition, HPF 1.0, was released in May 1993 [24]. Requests for corrections, clarifications, and interpretations to the HPF 1.0 version as well as the Fortran 95 standard resulted in version HPF 2.0 [26].[1] The HPF Forum, a consortium from industry, government labs, and academia built to define an HPF standard, had two important goals in mind. On the one hand, fast compiler development by vendors shall be possible. On the other hand, advanced language features shall be provided. A compromise to satisfy both goals was to divide the language definition into two parts. The HPF 2.0 Language defines the minimal requirements which each HPF compiler has to meet. The base language contains only features which are expected to be efficiently implemented by compiler vendors within a short period of time. The HPF 2.0 Approved Extensions include advanced features that require more implementation effort and are not likely to be supported by compilers from the very beginning.

Similar to the predecessor languages, HPF provides alignment and distribution. In the base language, remappings may only occur at procedure calls and take place, if the mapping of the actual argument differs from the mapping described by the dummy argument. An overriding principle in the base language is that all objects accessible to the caller are mapped after the call exactly as they were before the call. Executable *realign-* and *redistribute*-directives are supported by the Approved Extensions only.

[1] Whenever the version number is omitted, we mean HPF 2.0.

7.2 Data Flow Analysis

E. Morel and C. Renvoise [65] proposed partial redundancy elimination based on bidirectional data flow systems for common subexpression elimination. The source of the bidirectional dependence is a safety criterion which prevents that a computation is hoisted to the entry of a node unless one of the following holds at the exit of each predecessor node m: (1) the computation can be hoisted to the exit of node m or, (2) the computation is available at the exit of node m.

D.M. Dhamdhere [17] modified the Morel-Renvoise algorithm by an edge placement approach. The safety criterion which led to the bidirectional dependence in the Morel-Renvoise algorithm was necessary because neither edge splitting was assumed as a prerequisite nor edge placement supported. With the edge placement algorithm as described in [17, 19] the safety criterion can be removed resulting in a unidirectional, backward data flow problem. However, the original Morel-Renvoise algorithm only hints imprecisely on profitability: partial availability is used as criterion to decide whether a particular code motion is profitable or not. This leads to unnecessary motions as well. In [19] a second profitability term has been introduced which hints on unnecessary code motions more accurately (but it is also only a heuristic). Unfortunately, this second profitability term introduces again bidirectional dependences. Further it is proposed to reduce the resolution costs of bidirectional equations by splitting them into a series of unidirectional data flows which are solved instead. Applying bidirectional partial redundancy elimination for assignments is discussed in [18]. Since interdependencies between assignments are possible, multiple runs of the algorithm may be necessary in order to get better results. But the profitability terms may prevent optimal results: an assignment which cannot be moved profitably may prevent a profitable motion of another assignment.

J. Knoop, O. Rüthing, and B. Steffen [52] present a lazy code motion strategy which places computations as early as necessary but as late as possible. Instead of using bidirectional data flows, it is realized by a sequence of unidirectional equations. Elimination of partially dead as well as partially redundant assignments is discussed in [53] and [54], respectively. There it is shown that both transformations can be solved optimally.

E. Morel and C. Renvoise [66] describe an interprocedural extension of the partial redundancy elimination algorithm presented in [65]. The interprocedural algorithm is performed as a two-pass mechanism. First, information is gathered for each procedure P which represents the impact of a call of P at the call site. Then, partial redundancy elimination is done by treating the procedures in reverse calling order. If the algorithm is applied in the presence of recursive procedures, approximations are made. The interprocedural extension is not a general interprocedural approach, but specifically developed for partial redundancy elimination.

J. Knoop and B. Steffen [56, 57] present a generic interprocedural data flow framework which can be used to solve arbitrary data flow systems. It is a generalization of the well-known intraprocedural Coincidence Theorem of Kam and Ullman [42]. The generalization covers imperative programs with recursive procedures, global and local variables, formal value and reference parameters.

Feigen et al. [22] presented an algorithm for eliminating partially dead assignments which moves even groups of statements in order to increase profitability. Contrary to other code motion systems, the branching structure is not preserved.

None of the techniques described above combine both partial redundancy and partial dead code elimination in one framework.

7.3 Remapping Analysis and DFA-based Communication Optimizations

M.W. Hall et al. [36] present techniques for hoisting remappings out of loops and eliminating dead remappings. But they do not consider the more general problem of eliminating partially dead and partially redundant remappings.

F. Coelho and C. Ancourt [15] 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.

D.J. Palermo, E.W. Hodges, and P. Banerjee [68] present several analyses related to dynamic remappings: computation of reaching mappings, making 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.

Communication optimization frameworks relying on code motion techniques for *partial redundancy elimination* (PRE) have been proposed by several researchers.

M. Gupta, E. Schonberg, and H. Srinivasan [32] use PRE-techniques for available section descriptors in order to reduce communication costs. With a single framework they perform communication vectorization, elimination of redundant communication, and latency hiding by moving communication earlier. Data flow analysis is based on the PRE-techniques developed by Morel/Renvoise [65] and Dhamdhere [19].

G. Agrawal, J. Saltz, and R. Das [1] focus on the problem of interprocedural placement of communication statements. They develop an interprocedural extension of the intraprocedural redundancy elimination approach of Dhamdhere [19]. The algorithm is also applicable to recursive procedures.

K. Kennedy and A. Sethi [46] propose a constraint-based communication placement framework based on the unidirectional PRE-algorithm for *lazy code motion* of [52]. It maximizes latency hiding by determining the earliest safe placement for sends and the latest balanced placement for the corresponding receives. Constrained communication buffer size is used to illustrate constraint analysis and constraint-based communication placement.

None of the techniques described above combine partial redundancy and partial dead code elimination.

7.4 Remapping Communication

The overall remapping runtime costs of a program depend on the number of executed remappings and the execution time required for individual remappings. Hence, efficient algorithms for migrating data elements to their new owning processors are essential. The development of efficient remapping routines for arrays has been addressed by several researchers.

R. Thakur et al. [79, 80] describe efficient single-phase array remappings from $block(m)$ to *cyclic*, *cyclic* to $block(m)$, and $cyclic(ky)$ to $cyclic(y)$ as well as $cyclic(x)$ to $cyclic(kx)$. Based on these remappings a two-phase approach is proposed for the general case $cyclic(x)$ to $cyclic(y)$ when there is no particular relation between x and y. The intermediate mapping m can be either the *greatest common divisor* or *least common multiple* of x and y, i.e. $m = gcd(x, y)$ or $m = lcm(x, y)$.

S.D. Kaushik et al. [43, 44, 45] proposed a general multi-phase remapping approach. The idea of the multi-phase approach is to perform a remapping by an appropriate sequence of remappings such that the communication costs of the sequence are less than that of direct remapping. Basically, the multi-phase strategy trades off a decrease in the number of message startups with an increase in the data volume transmitted. Based on an algebraic representation of regular data mappings, a cost model for estimating the communication overhead of remappings is developed. Using this model,it is shown that the multi-phase strategy can reduce the total cost for array remappings.

S. Ramaswamy et al. [70, 71, 72] presents a new mathematical representation for regular mappings called PITFALLS (Processor Index Tagged FAmily of Line Segments). Based on this representation remapping algorithms are discussed. One of the significant contributions is being able to handle arbitrary source and target processor sets while performing remappings. Also an arbitrary number of array dimensions can be handled in a scalable manner.

Y.-C. Chung et al. [13] present a basic-cycle calculation technique to efficiently perform a $cyclic(s)$ to $cyclic(t)$ remapping. The idea is to compute for some specific array elements in a basic-cycle, which is defined by $lcm(s, t)/gcd(s, t)$, the communication sets. The communication sets of the basic-cycle are used to efficiently perform the remapping of the whole array.

E.T. Kalns and L.M. Ni [40] proposed a processor mapping technique to minimize the total amount of data that must be communicated during remapping. This issue has also been addressed in [15] (cf. to Section 7.3).

A. Wakatani and M. Wolfe [81, 82] proposed a strip-mining approach for remapping instead of transferring the entire array at one time in order to reduce the communication overhead by latency hiding. The idea of message strip-mining is to restructure the code such that the communication overhead is overlapped with the subsequent computation. This is achieved by dividing communication and computation into several blocks and performing them in a pipeline manner.

Chapter 8

Conclusions

In this thesis we have presented analyses and transformations for reducing the number of remappings executed at runtime. In order to make classical analyses amenable for remapping analyses and to allow a uniform handling of all sources of remappings, we introduced mapping assignments. Basically, our approach evolves from combining partial dead code elimination PDCE and partial redundancy elimination PRAE. Contrary to the component transformations PDCE and PRAE for which a "globally best" program exists, a "globally best" program does not exist in general any more for MAP. A hierarchy of algorithms for mapping assignment placement of varying power and efficiency offering customized solutions according to a user's requirements and preferences is presented. 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 MAP-algorithm.

8.1 Contributions

The primary contributions of this thesis are summarized below.

- An intermediate representation based on mapping assignments has been introduced which simplifies remapping analyses considerably. The insertion of mapping assignments is done by the compiler in an initial phase and discussed in detail. Above all the two-level mapping makes insertion of mapping assignments complicated.

- Based on the ideas of bidirectional partial redundancy elimination, bidirectional partial dead code elimination has been developed. A bidirectional mapping assignment placement framework combines both bidirectional partial redundancy and partial dead code elimination.

- The MAP approach which is based on PDCE and PRAE is developed and its optimality discussed. It is shown that an optimal solution does not exist in general any more. Finally, we arrive at hierarchy of MAP-algorithms of varying power and efficiency offering customized solutions according to a user's requirements and preferences. Note that the

MAP approach and the optimality results are not restricted to mapping assignments, but may be interesting also in other contexts.

- We extended MAP interprocedurally and presented the anomalies resulting from a naive extension. A constraint on motion of assignment patterns suppressed the anomalies and allowed a uniform handling of both mapping assignments and ordinary assignments. Again, we arrive at a hierarchy of IMAP-algorithms.

8.2 Future Research

In our approach we perform code motion in order to reduce the number of executed mapping assignments. Code motion can help to reduce the communication overhead of individual remappings as well. Basically, communication costs are characterized by the start-up time, the time required to copy data from the user space to system space and vice versa, and the time needed for the actual data transfer. By splitting a remapping into a sending and receiving component and overlapping communication with computation, the transfer time as well as the time required for copying data can be hidden and the performance improved considerably. But note that there are tradeoffs between communication latency hiding and reducing the number and volume of messages (e.g. by message coalescing and aggregating) which have to be considered by communication placement strategies [20, 21].

Further, our optimality considerations have to be refined. Currently, we conjecture that like in the example of Figure 4.12 "maximally transformed" programs are either identical (up to irrelevant reorderings in basic blocks) or incomparable with respect to the order of Section 4.4.3 counting assignments for every assignment pattern separately. This point has to be clarified.

Finally, path profiling [33] has to be considered which allows to distinguish between frequently and less frequently executed paths which can be used to guide mapping assignment placement.

Bibliography

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

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

[3] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19:26–34, 1986.

[4] A. Ayers, R. Gottlieb, and R. Schooler. Aggressive inlining. In *Proc. of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI'97)*, pages 134–145, Las Vegas, Nevada, 1997.

[5] M. Baber. Hypertasking: Automatic array and loop partitioning on the iPSC. In *Proc. of the Hawaii International Conference on System Sciences, Software Track, Issues in Parallel Programming*, Koloa, Hawaii, January 1991.

[6] M. Baber. Hypertasking Support for Dynamically Redistributable and Resizeable Arrays on the iPSC. In *Proc. of the Sixth Distributed Memory Computing Conference*, pages 59–66, Portland, Oregon, April/May 1991.

[7] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Orca: A Language For Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3), March 1992.

[8] S. Barros, D. Dent, L. Isaksen, G. Robinson, G. Mozdzynski, and F. Wollenweber. The IFS model: A parallel production weather code. *Parallel Computing*, 21, 1995.

[9] 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.

[10] S. Benkner, B. Chapman, and H. Zima. Vienna Fortran 90. In *Proceedings of the SHPCC Conference 1992*, Williamsburg,VA, 1992.

[11] K.M. Chandy and C. Kesselman. Compositional C++: Compositional Parallel Programming. In *Languages and Compilers for Parallel Computing, 5th International Workshop*, Lecture Notes in Computer Science, Vol. 757, pages 124–144. Springer Verlag, August 1992.

[12] B. Chapman, H. Zima, M. Haines, P. Mehrotra, and J. Van Rosendale. OPUS: A Coordination Language for Multidisciplinary Applications. Technical Report TR 95-6, Institute for Software Technology and Parallel Systems, University of Vienna, Vienna, Austria, Oktober 1995.

[13] Y.-C. Chung, C.-H. Hsu, and S.-W. Bai. A basic-cycle calculation technique for efficient dynamic data redistribution. *IEEE Transactions on Parallel and Distributed Systems*, 9(4), April 1998.

[14] J. Clinckemaillie, B. Elsner, G. Lonsdale, S. Meliciani, S. Vlachoutsis, F. de Bruyne, and M. Holzner. Performance issues of the parallel PAM-CRASH code. *Journal of Supercomputing Applications and High-Performance Computing*, 11(1), 1997.

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

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

[17] D.M. Dhamdhere. A Fast Algorithm for Code Movement Optimization. *ACM SIGPLAN Notices*, 23(10):172–180, 1988.

[18] D.M. Dhamdhere. Practical Adaption of the Global Optimization Algorithm of Morel and Renvoise. *ACM Transactions on Programming Languages and Systems*, 13(2):291–294, April 1991.

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

[20] T. Fahringer and E. Mehofer. Buffer-safe communication optimization based on data flow analysis and performance prediction. In *Proc. of the IEEE Int'l Conference on Parallel Architectures and Compilation Techniques* (*PACT'97*), pages 189 – 200, San Francisco, CA, November 1997.

[21] T. Fahringer and E. Mehofer. Problem and machine sensitive communication optimization. In *Proc. of the ACM SIGARCH Int'l Conference on Supercomputing* (*ICS'98*), Melbourne, Australia, July 1998. To appear.

[22] L. Feigen, D. Klappholz, R. Casazza, and X. Xue. The revival transformation. In *Proc. of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'94)*, pages 421–434, Portland, Oregon, January 1994.

[23] M.J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, 1972.

[24] High Performance Fortran Forum. High Performance Fortran language specification version 1.0. *Scientific Programming*, 2(1-2):1–170, Spring and Summer 1993. Also available as Technical Report, Rice University, Houston, TX, May 3, 1993.

[25] 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.

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

[27] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, July 1997.

[28] I. Foster and K.M. Chandy. Fortran M: A language for modular parallel programming. *Journal of Parallel and Distributed Computing*, 26, 1995.

[29] I. Foster and S. Taylor. *Strand: New concepts in parallel programming.* Prentice-Hall, Englewood Cliffs, 1990.

[30] 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.

[31] 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.

[32] 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.

[33] R. Gupta, D. Berson, and J. Fang. Path profile guided partial dead code elimination using predication. In *Proc. of the IEEE Int'l Conference on Parallel Architectures and Compilation Techniques* (*PACT'97*), pages 102 – 113, San Francisco, CA, November 1997.

[34] S.K.S. Gupta, C.-H. Huang, and P. Sadayappan. Implementing Fast Fourier Transforms on distributed-memory multiprocessors using data redistributions. *Parallel Processing Letters*, 4(4):477–488, December 1994.

[35] M. Hall. *Managing Interprocedural Optimization.* PhD thesis, Rice University, Houston,TX, April 1991.

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

[37] K.A. Hawick, E.A. Bogucz, A.T. Degani, G.C. Fox, and G. Robinson. High Performance Fortran for computational fluid dynamics applications. Technical Report NPAC TR SCCS-638, Northeast Parallel Architectures Center, Syracuse University, Syracuse, NY, September 1994.

[38] M.S. Hecht. *Flow Analysis of Computer Programs.* Programming Language Series. North-Holland, 1977.

[39] R. Hempel, H.-C. Hoppe, U. Keller, and W. Krotz. *PARMACS V6.0 Specification.* PALLAS G.m.b.H., D-50321 Bruehl, Germany, August 1994.

[40] E.T. Kalns and L.M. Ni. Processor mapping technique toward efficient data redistribution. In *Proc. of the International Parallel Processing Symposium*, pages 469–476, April 1994.

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

[42] J.B. Kam and J.D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.

[43] S.D. Kaushik, C.-H. Huang, R.W. Johnson, and P. Sadayappan. An approach to communication-efficient data redistribution. In *Proc. of the 1994 ACM International Conference on Supercomputing*, pages 364–373, Manchester,England, July 1994.

[44] S.D. Kaushik, C.H. Huang, J. Ramanujam, and P. Sadayappan. Multi-phase Array Redistribution: A Communication Efficient Approach to Array Redistribution. Technical Report OSU-CISRC-9/94-TR52, Ohio State University, Columbus,OH, 1994.

[45] S.D. Kaushik, C.H. Huang, J. Ramanujam, and P. Sadayappan. Multi-phase array redistribution: Modeling and evaluation. In *Proc. of the International Parallel Processing Symposium*, 1995.

[46] 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.

[47] G. Kildall. A unified approach to global program optimization. In *Proc. of the ACM Symposium on Principles of Programming Languages*, pages 194–206, 1973.

[48] J. Knoop. *Optimal interprocedural program optimization: A new framework and its application.* PhD thesis, University of Kiel, Germany, 1993.

[49] J. Knoop and E. Mehofer. Interprocedural distribution assignment placement: More than just enhancing intraprocedural placing techniques. In *Proc. of the IEEE Int'l Conference on Parallel Architectures and Compilation Techniques* (*PACT'97*), pages 26 – 37, San Francisco, CA, November 1997.

[50] J. Knoop and E. Mehofer. Intraprocedural Distribution assignment placement: Optimality and flexibility. Technical Report TR 97-6, Institute for Software Technology and Parallel Systems, University of Vienna, Austria, 1997.

[51] J. Knoop and E. Mehofer. Optimal distribution assignment placement. In *Proc. of Euro-Par'97*, Lecture Notes in Computer Science, Vol. 1300, pages 364 – 373, Passau, Germany, August 1997. Springer-Verlag, Heidelberg, Germany.

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

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

[54] 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.

[55] 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.

[56] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *Proc. of the 4th Conference on Compiler Construction*, Paderborn, Germany, 1992.

[57] J. Knoop and B. Steffen. Efficient and optimal bit-vector data flow analyses: A uniform interprocedural framework. Technical Report Nr. 9309, Christian-Albrechts-Universität zu Kiel, Kiel,Germany, 1993.

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

[59] C. Koelbel. *Compiling Programs for Nonshared Memory Machines.* PhD thesis, Purdue University, West Lafayette,IN, August 1990.

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

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

[62] P. Mehrotra and J. Van Rosendale. The BLAZE language: A parallel language for scientific programming. *Parallel Computing*, 5:339–361, 1987.

[63] P. Mehrotra and J. Van Rosendale. Programming distributed memory architectures using Kali. In A. Nicolau, D. Gelernter, T. Gross, and D. Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, pages 364–384. Pitman/MIT-Press, 1991.

[64] K. Mills, G. Cheng, M. Vinson, S. Ranka, and G. Fox. Software issues and performance of a stock option pricing model on the Connection Machine-2 and DECmpp-12000. In *Proceedings of the 5th Australian Supercomputing Conference*, Melbourne, Australia, December 1992.

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

[66] E. Morel and C. Renvoise. Interprocedural Elimination of Partial Redundancies. In S.S Muchnick and N.D. Jones, editors, *Program Flow Analysis. Theory and Applications*, chapter 6. Englewood Cliffs NJ: Prentice-Hall, 1981.

[67] S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers, San Francisco, CA, 1997.

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

[69] L. Prylli and Tourancheau. Efficient block-cyclic data redistribution. In *Proc. of Euro-Par'96*, Lecture Notes in Computer Science, Vol. 1123, pages 155–164, Lyon, France, August 1996. Springer-Verlag, Heidelberg, Germany.

[70] S. Ramaswamy and P. Banerjee. Automatic generation of efficient array redistribution routines for distributed memory multicomputers. Technical Report UILU-ENG-94-2213,CRHC-94-09, Center for Reliable and High Performance Computing, Computer Systems and Research Laboratory, Urbana,IL 61801, April 1994.

[71] 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, February 1995.

[72] 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.

[73] S.E. Richardson and M. Ganapathi. Interprocedural analysis vs. procedure integration. *Information Processing Letters*, 32:137–142, August 1989.

[74] 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.

[75] K. Schwarz and P. Blaha. Description of an LAPW DF program (WIEN95). *Lecture Notes in Chemistry*, 67, 1996.

[76] J. Subhlok, D.R. O'Hallaron, and T. Gross. Task parallel programming in Fx. Technical Report CMU-CS-94-112, School of Cumputer Science, Carnegie Mellon University, Pittsburgh, PA, August 1994.

[77] J. Subhlok, J. M. Stichnoth, D. R. O'Hallaron, and T. Gross. Exploiting Task and Data Parallelism on a Multicomputer. In *Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 13–22, San Diego,CA, May 1993.

[78] V. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, 1990.

[79] R. Thakur, A. Choudhary, and G. Fox. Runtime Array Redistribution in HPF Programs. In *Proc. of the Scalable High-Performance Computing Conference*, pages 309–316, Knoxville, Tennessee, May 1994.

[80] R. Thakur, A. Choudhary, and J. Ramanujam. Efficient algorithms for array redistribution. *IEEE Transactions on Parallel and Distributed Systems*, 7(6):587–594, June 1996.

[81] A. Wakatani and M. Wolfe. A new approach to array redistribution: Strip mining redistribution. In *Proc. of Parallel Architecture and Languages Europe*, July 1994.

[82] A. Wakatani and M. Wolfe. Effectiveness of message strip-mining for regular and irregular communication. In *Proc. of Seventh Int'l Conf. on Parallel and Distributed Computing Systems*, October 1994.

[83] 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.

[84] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press Frontier Series. Addison-Wesley, 1990.

Appendix A

Reaching Align/Realign Directives

The problem of finding the set of possibly aligned arrays is identical to the problem of finding all ALIGN/REALIGN directives that are reaching some REDISTRIBUTE directive (cf. Section 3.3.2). Table A.1 summarizes the data flow analyses for reaching alignments.

Reaching Alignment Analysis (RA): (Let $\alpha \in \mathcal{A}$)

Local predicates: $N \times \mathcal{A} \rightarrow \{true, false\}$

- $\mathbf{LocRA}_n(\alpha)$: There is an alignment α in n which is valid at the exit of n.
- $\mathbf{LocBlock}_n(\alpha)$: The alignment α is blocked by some instruction of n.
- $\mathbf{LocSpec}(\alpha)$: There is an alignment α in the specification statements of the corresponding scoping unit.

Data flow equations: [1]

$$
\begin{aligned}
\textbf{N-RA}_n(\alpha) &= \begin{cases} \mathbf{LocSpec}(\alpha) & \textit{if} \quad n = \mathbf{s} \\ \sum\limits_{m \in pred(n)} \textbf{X-RA}_m(\alpha) & \textit{otherwise} \end{cases} \\
\textbf{X-RA}_n(\alpha) &= \mathbf{LocRA}_n(\alpha) + \textbf{N-RA}_n(\alpha) * \overline{\mathbf{LocBlock}_n(\alpha)}
\end{aligned}
$$

Key:

[1] Boolean conjunctions are denoted by $*$ and Π, Boolean disjunctions by $+$ and Σ, and Boolean negation is denoted by a bar (e.g. $\overline{\mathrm{A}}$).

Table A.1: Reaching Alignment Analysis

For basic definitions and terminology refer to Section 4.1. Let us consider a flow graph G and let $\mathcal{A}$ denote the set of ALIGN and REALIGN directives occurring in G. Intuitively, **LocRA** denotes all REALIGN directives which occur in n and are not followed by any other REALIGN directive in n affecting the same alignee. A REALIGN is blocked by n (specified by **LocBlock**) if a REALIGN occurs in n with the same alignee as the REALIGN under consideration. For start node **s** we have to take ALIGN directives occurring in the specification statements into account which is achieved by initializing **N-RA** for the start node with **LocSpec**. Note that REDISTRIBUTE directives do not affect reaching alignment analysis. Let $\mathbf{N\text{-}RA}^\star$ denote the solution of the equation systems for reaching analysis. Then $\mathbf{N\text{-}RA}^\star$ can be used as basis for inserting mapping assignments at REDISTRIBUTE directives in order to remap the set of aligned arrays.

Appendix B

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 B.1. The first application of PDCE eliminates map(A) := CYCLIC(i) at node **6**, which turns the assignment map(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 B.3. These eliminations suspend the blockades of the partially dead occurrences of map(B) := $\alpha_{\mathsf{A}}^{\mathsf{B}}$ at nodes **12** and **26**, which can now be eliminated by a second application of PDCE resulting in Figure B.4. As a consequence, map(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 B.6 which is stable under further admissible assignment motions and dead (redundant) assignment eliminations.

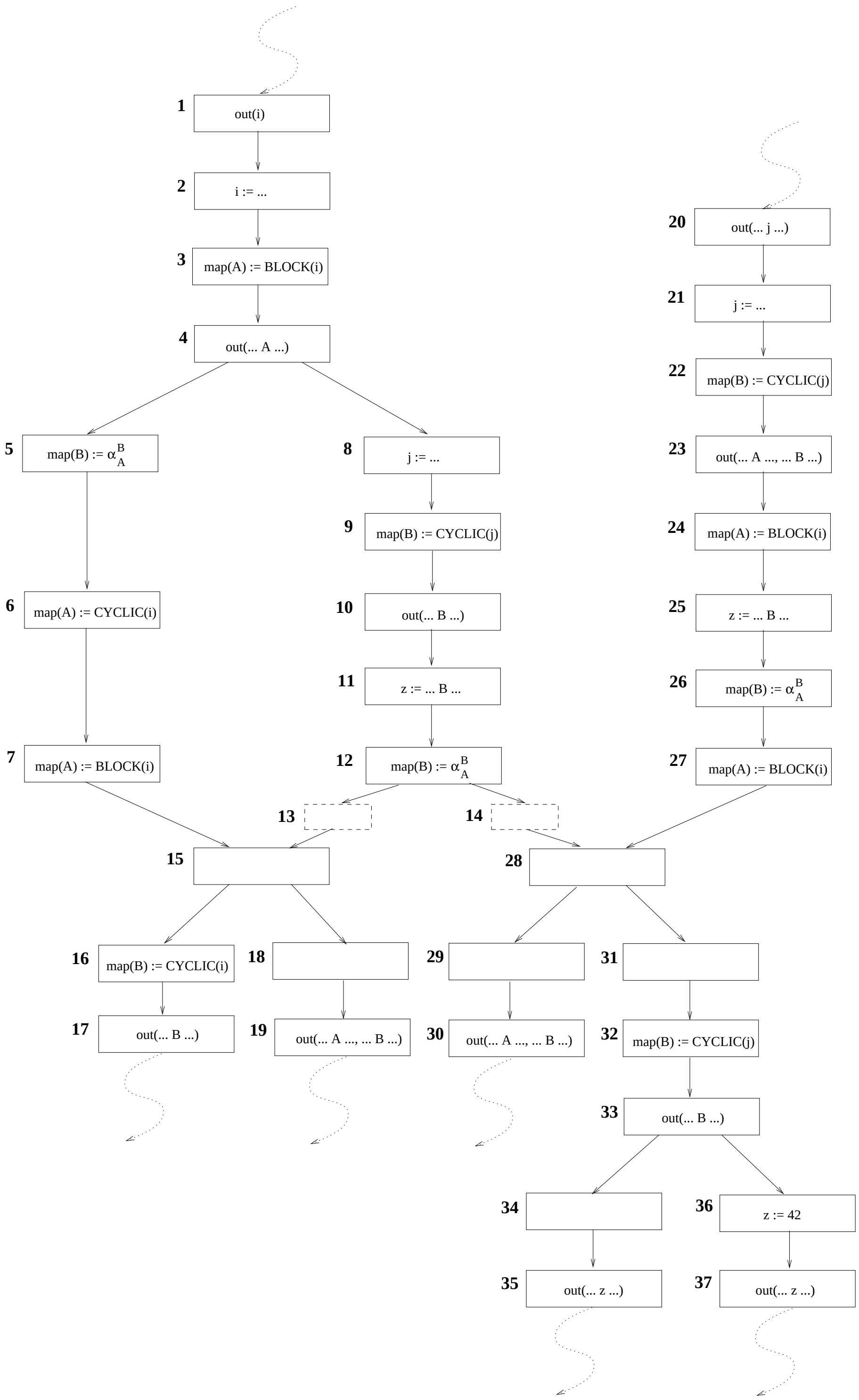


Figure B.1: The Original Flow Graph

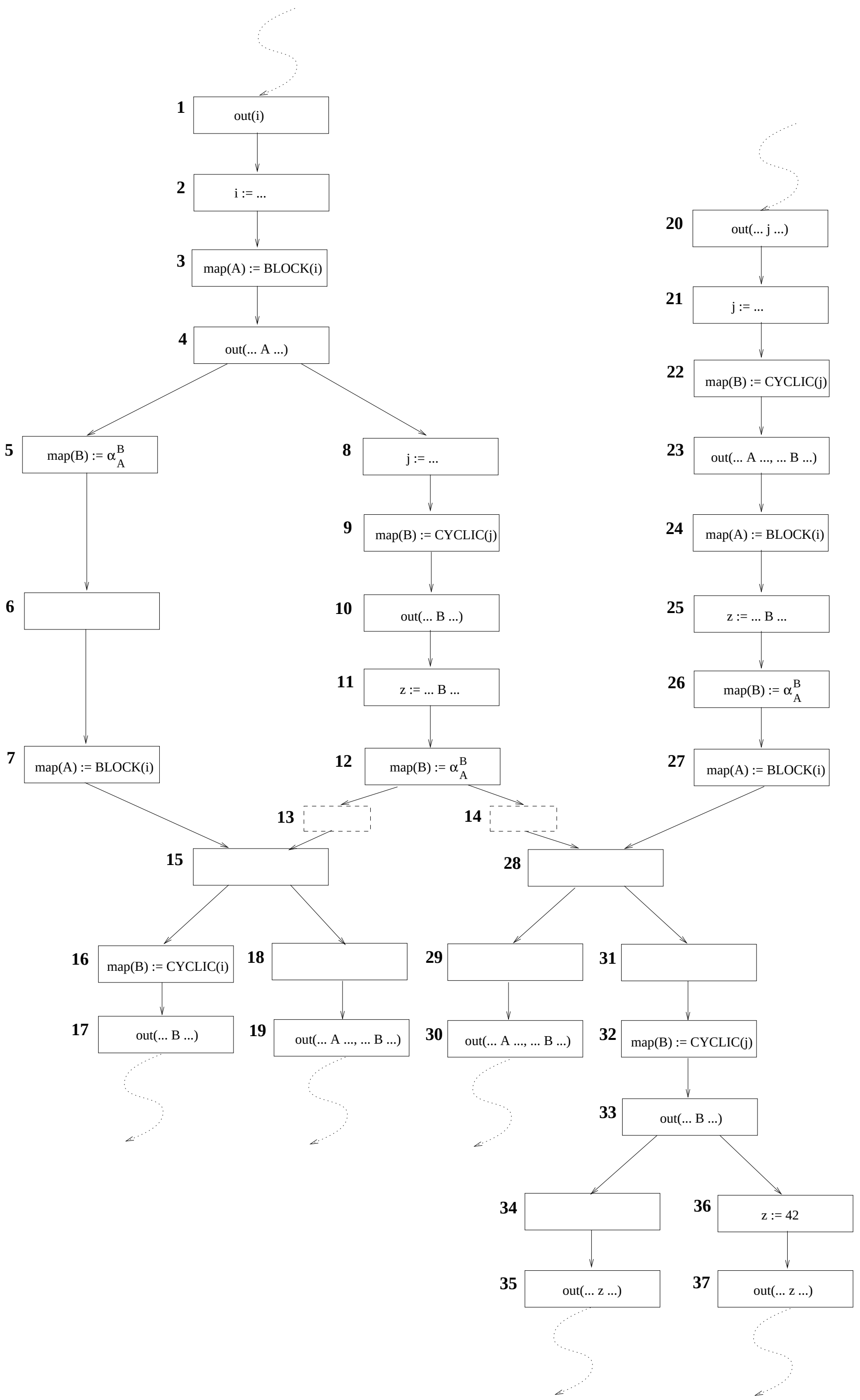


Figure B.2: After PDCE

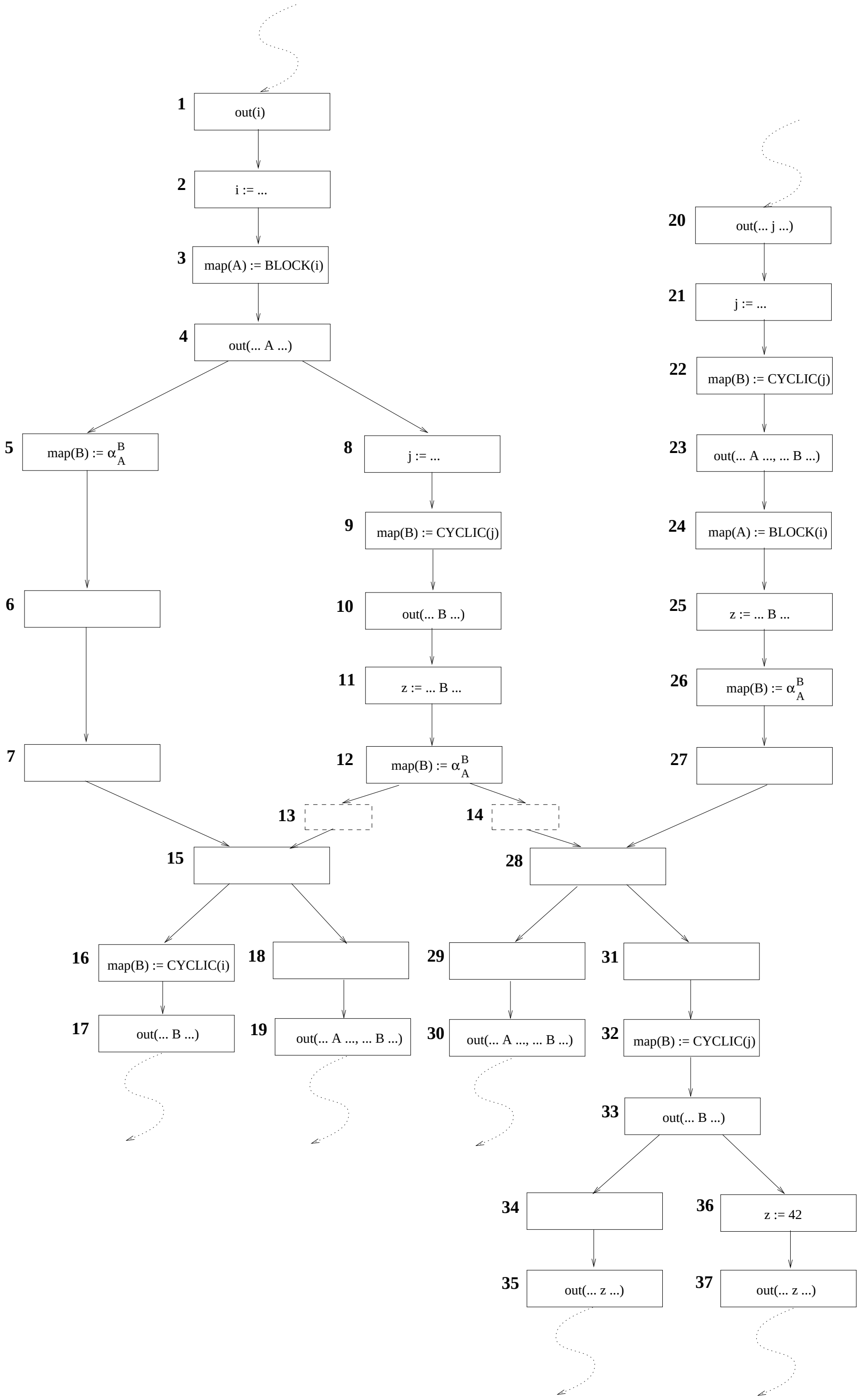


Figure B.3: After PDCE - PRAE

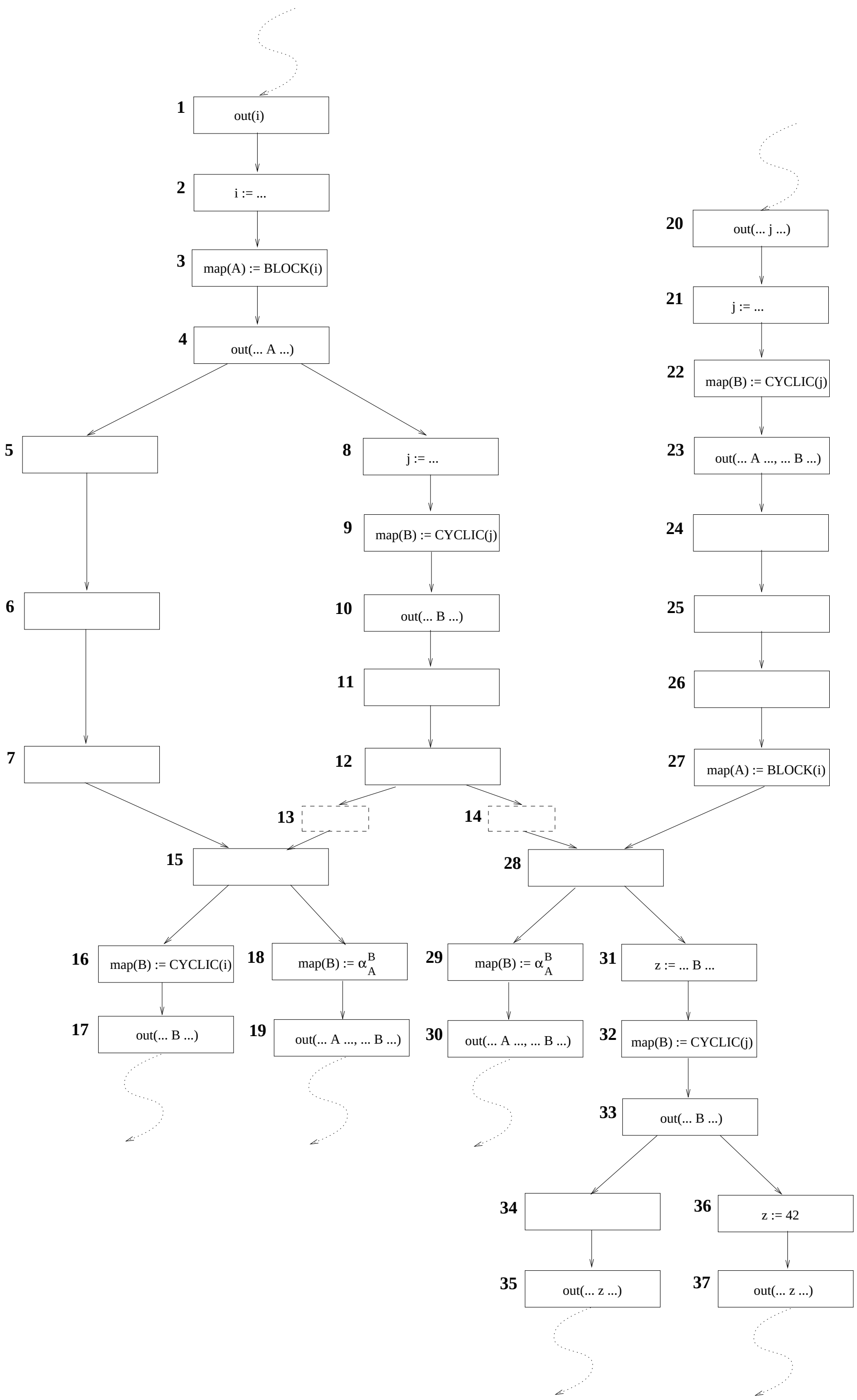


Figure B.4: After PDCE - PRAE - PDCE

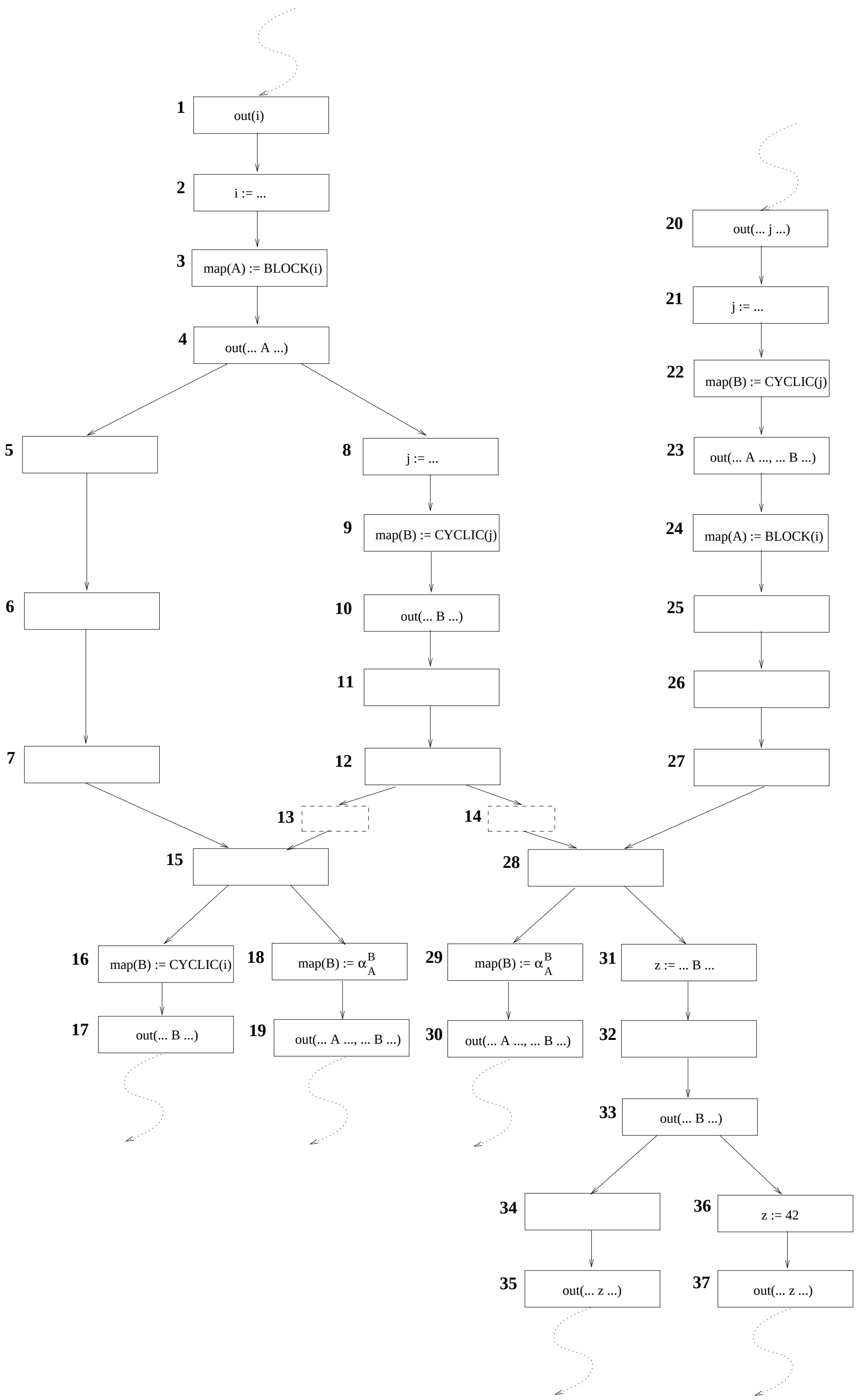


Figure B.5: After PDCE - PRAE - PDCE - PRAE

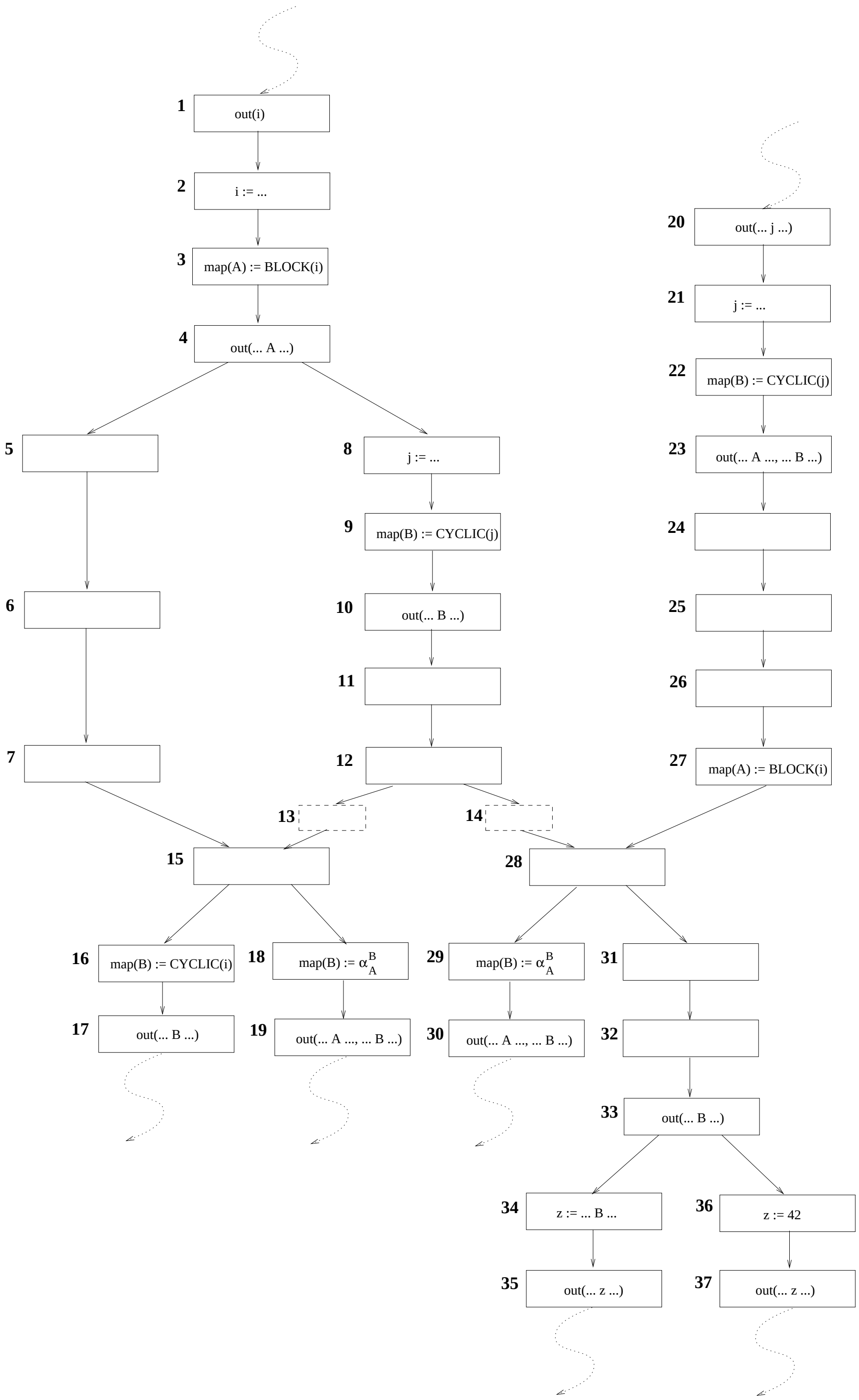


Figure B.6: Stable after PDCE - PRAE - PDCE - PRAE - PDCE