UNIVERSITÄT PASSAU
Fakultät für Mathematik und Informatik

Dissertation

The Mechanical Parallelization of Loop Nests Containing while Loops

Author:
Martin Griebl

Advisor:
Prof. Christian Lengauer Ph. D.

October 15, 1996

Acknowledgments

Nobody can write a thesis without help form others, and it is usually impossible to express one's gratitude for this immense amount of help. The least I can do is to devote the first pages of my thesis to all these wonderful people, and thank them all for their precious support and individual help.

I want to mention some people explicitly, even knowing that my list must be inclomplete.

First of all, I want to thank Professor Christian Lengauer who has been an excellent advisor to me. Thank you for my position, for your liberality concerning working modes, for uncountably many fruitful discussions with you (official and private), for your indefatigability in improving my English, for multiple detailed proof readings of this thesis, for always having time for my problems, ...; short, thank you for having been a real "Doktorvater", which, to me, is more than an advisor.

In addition, I am grateful to Professor Paul Feautrier: thank you that you have accepted to review this thesis, and took the time to give me detailed comments on my draft of this thesis.

I also want to thank Professor P. Kleinschmidt, Professor F.-J. Brandenburg and Professor W. Hahn for having agreed to being my examiners and for their helpfulness. Furthermore, I would like to thank Professor N. Schwartz who always helps at the formal aspects on the way to a Ph. D.

However, there are also helpful persons outside of my dissertation committee. First of all, I want to thank my French colleague and friend Jean-François Collard. Thank you for your cooperation already at the beginning of this thesis, when we did not yet know each other. Because of your open-minded way, we succeeded in working together instead of being competitors. This led to many fruitful discussions and a deep friendship. Thanks a lot for that.

Furthermore, I would like to thank the members of the Lehrstuhl für Programmierung for the good working climate and for some helpful hints, and esp. Christoph Herrmann for his excellent proof reading. In addition, there is another member of the group I want to mention specifically: Ulrike Lechner. I would call her "the good soul of our group". Thank you for sharing the office and some work, and for the wonderful climate in our office, not only due to your flowers.

A-pro-pos climate: one of the most agreeable teams I have ever been part of is the LooPo team. The students in this team have been a continuous source of energy and encouragement to me. Numerous discussions helped me understand the problems in the various facets of loop parallelization. I am grateful to Andreas Dischinger, Peter Faber, Robert Günz, Harald Keimer, Radko Kubias, Wolfgang Meisl, Frank Schuler, Martina Schumergruber, Sabine Wetzel, Christian Wieninger and Alexander Wüst. A specific thank is due to Nils Ellmenreich:

thank you, for being a co-leader of LooPo, and still more for your unbounded helpfulness and for your friendship.

Right, there is one name missing in the LooPo team: Max Geigl. You can be sure that I have not forgotten you, Mäx; I just want to thank you separately. Through how many long drives did you have to listen to and discuss with me about code generation schemes or more strange things like multi-dimensional combs? Thank you for never jumping off the car, and also for accepting that other people around us called us crazy because of our "vacuum cleaner stories". More seriously, thank you for always having time for me, in short, thank you for being a really good friend.

From the university, I want to thank in addition our secretaries, Johanna Bucur and Ulrike Peiker who kept administrative work as far away from me as possible. Similarly, my friend and colleague Andreas Stübinger and our student members of the staff, Sven Anders, Holger Bischof, and Bernhard Lehner skilled me from a lot of system administration and implementation work—thanks to all of you.

However, there is not only the professional support necessary for success. Almost more importantly, one needs an environment that radiates safety and that provides one with new energy. This environment has always been my family. Thanks a lot for that. Unfortunately, precisely the same people have to stand aside when work requires more time. I want to thank my parents and my wife for understanding and accepting this. Thank you, Gabi, for fighting hard to understand what I am working on, and for trying to help me. Thanks for guarding me from all those things which I had no energy for—you could not have done more.

Contents

Chapter 1

Introduction

Technological advances in the last decades have led to faster and faster computer systems, but the demands made on the speed of computer systems are growing just as rapidly. Large computational problems are becoming so data-intensive that sequential systems, i.e., systems with only one main processor often have not enough power to solve these problems in the time required by the user.

This has led to the development of parallel computers, i.e., systems with more than one main processor. The crucial problem posed by these systems is how to write programs for them: one can either re-implement existing sequential algorithms so as to adjust them for multi-processor computers or redesign algorithms for parallel systems from scratch. Both approaches have one common disadvantage: they are costly and error-prone if done by hand.

Consequently, much effort has been invested in research of how to transform automatically sequential programs into programs for multi-processor systems. This has led to the emergence of the research area of *automatic parallelization*. For several reasons there has been a focus on nested loops: first, many programs spend the main part of their execution time in loop nests—this makes loop parallelization worth-while; second, the amount of potential parallelism in loop nests turns out to be considerable—orders of magnitude of speedup are possible; third, the regularity of many loop nests facilitates the automatic detection of parallelism and has aided the development of efficient parallelization techniques.

Basically, there are two different approaches to loop parallelization: an *experimental* and a *model-based approach*. In the experimental approach a set of possible loop transformations has been developed among which one can/must choose useful ones heuristically if one wants to parallelize a concrete program; this approach led to first good results.

The other approach is based on a mathematical model. In order to develop a clean model, it is usually impractical to consider programs immediately as they occur in general applications. Instead one first considers a subset of "well-behaved" applications, for which a model can be developed more easily. Then, one tries to relax some of the restrictions and thereby make the model more complex and general. This has also been done in loop parallelization.

Mainly, three restrictions have aided the development of a computational model for loop parallelization. First, in typical programming languages there is a general type of loops, while *loops*, and a more restricted type, for *loops*. The main difference is that in for loops the number of iterations is known at compile time (or, at the latest, when the loop starts execution) whereas in while loops it is not. It turned out that while loops—and even arbitrary for loops—are still too general for the development of a simple mathematical model, but it was possible to find a model for nested for loops whose bounds are affine expressions in outer loop indices and *structure parameters*, i.e., symbolic constants. We call such loops *affine loops*.

Second, orthogonally to the first point, also the nesting order influences the development of a simple model: in the restricted case of *perfect loop nests* only the innermost loop contains statements different from loops; this is not true for the general case of *imperfect loop nests*.

Third, there can be arbitrary dependences between the computations spawned by a loop nest. In order to model these dependences, they should be *uniform*, i.e., identical for all computations, or at least *affine*, i.e., affine functions in the loop indices (more precise definitions are given in Section 3.1.1).

The first mathematical model was developed for perfect nests of affine loops with uniform dependences: it is called the *polytope model* [40]. In brief, polytopes are finite convex geometrical objects with plane borders. Mathematically, they are bounded *polyhedra*, where a polyhedron is a finite intersection of halfspaces. The exact correspondence between polytopes and loop nests is explained in the next chapter. The existing generalizations of the polytope model are described in Section 2.1.

As just noted, the polytope model for the parallelization of loop nests has a more restricted range of application than the experimental approach; on the other hand, it supports parallelization methods which are fully automatic and—within the choices offered by the model—provably optimal.

Currently, one can observe a convergence of both approaches; the model the second approach is based on is being extended such that the techniques of the first approach can be expressed, and many of the restrictions formerly necessary have been relaxed.

Before work on this thesis began, the parallelization methods of both the model-based approach and the experimental approach did not support the detection of any parallelism hidden in a nest of general loops, even if there are only affine dependences. The contribution of this thesis is a generalization of the polytope model to support the automatic parallelization of general loop nests, as long as their dependences are affine. We focus on the theoretical extensions of existing methods.

However, we also address the implementation of the extended methods in our work. For this purpose, we are developing *LooPo*, a source-to-source parallelization framework in which various well-known methods of loop parallelization in the polytope model are implemented. LooPo's extension to general loop nests, however, is ongoing work and, therefore, LooPo is not a focus of this thesis.

Loop nests containing while loops and for loops with arbitrary bounds occur frequently, e.g., in algorithms for sparse data structures. Thus, they are a major field of application of our parallelization methods.

Our approach also covers convergent iterative algorithms, frequent in numerical applications, which are usually while loops. However, these loops have special properties (cf. Section 4.2) whose exploitation is not a focus of this thesis; our goal is to develop a parallelization method that is generally applicable.

The thesis is organized as follows. Chapter 2 gives an overview of related work, terminology and the parallelization in the polytope model, and presents an example application which is used throughout this thesis. Chapter 3 presents in more detail the most important stages of a parallelization in the polytope model and analyzes, for every stage, the extensions that are necessary to integrate while loops. Chapter 4 offers a classification of loops which determines for every loop in a source nest how it is modeled and how it is treated during code generation. The subsequent parts of this thesis are more technical and deal with the irregularities which are introduced into the extended model due to the limited information available on the bounds of while loops: Chapter 5 tackles irregularities inside the target domain and Chapters 6 and 7 deal with the detection of the bounds of the target domain, Chapter 6 for dimensions in space and Chapter 7 for dimensions in time. Chapter 8 describes the current state of our source-to-source parallelizer LooPo. Finally, Chapter 9 concludes the thesis and discusses future work.

Chapter 2

Overview

We describe first the state of the art in loop parallelization and present our notation and some necessary definitions. Then, we specify the input required and the output supplied by our methods. Subsequently, the model is presented including all necessary extensions and all steps of the parallelization method are described briefly. Finally, we introduce a loop program which is used as an example throughout the thesis.

2.1 Related Work

The polytope model enables the parallelization of perfectly nested affine loops. The seminal work on the polytope model was done by Karp, Miller and Winograd [36] thirty years ago; it offers a way of scheduling systems of uniform recurrence equations. In 1974, Lamport [39] applied these ideas to loop nests and gave an algorithm for scheduling the iterations of a perfect nest of affine loops.

In the last two decades the methods of the polytope model have been extended in various directions, e.g., more precise dependence analysis techniques have been developed [28] and more flexible transformations [65] or by-statement transformations [19, 37, 53] (cf. Section 2.4.1) have been introduced.

However, a relaxation of the serious restriction of the affinity of the loop bounds was not considered before work on this thesis began. As we shall see in the mathematical definitions, such a relaxation transcends the framework of polytopes.

The parallelization of while loops has been investigated for a number of years [8, 59, 62, 64]. The general approach has been to pipeline the successive iterations where possible (e.g., [59, 64]). This does not require methods based on the polytope model, and it yields at most constant speedup.

Other approaches [62, 64] present *specific cases* in which the parallelization of while loops is possible, esp. for while loops which are actually disguised for loops. But none of these approaches offers a way of parallelizing nests with while loops in the *general case*, even if there exists potential parallelism.

The common problem of all these attempts is that they try to parallelize every while loop in a loop nest in isolation. This is, in general, impossible since the semantics of while is inherently sequential. However, in a nest of while loops considered as a whole one can detect and exploit parallelism.

We shall see that our approach subsumes the pipelining methods as well as parallelization possibilities in the specific cases of [64].

Up to now, our approach has also been used in the methods of J.-F. Collard and P. Feautrier who concentrate on the data dependence analysis in the extended model [16] and apply *speculative execution* [15], i.e., they allow that some statement S in the body of a source loop nest iterates farther in the target program than in the source program. If the additional iterations of S produce undesired values the proper final values must be recovered. This leads to serious problems in code generation. Thus, we choose the more restrictive *conservative execution* scheme which forbids additional iterations of S in the target program.

In this thesis we concentrate on the extensions of the polytope model and its methods and on the generation of target programs in the extended model, and we apply the results of Collard and Feautrier where we need them.

2.2 Mathematical Definitions and Notations

Our mathematical notation follows Dijkstra [24]. Quantification over a dummy variable x is written $(Q\ x : R.x : P.x)$. Q is the quantifier, R is a predicate in x representing the range, and P is a term that depends on x. Formal logical deductions are given in the form:

$$\begin{array}{ll} & \mathit{formula}_1 \\ op & \{ \text{ comment explaining the validity of relation } op \ \} \\ & \mathit{formula}_2 \end{array}$$

where op is an operator from the set $\{\Leftarrow, \Leftrightarrow, \Rightarrow\}$. The boolean values true and false are denoted by *tt* and *ff*, respectively.

The dimension of a vector $\vec{x}$ is denoted by $|\vec{x}|$. The projection to its coordinates k to l is written as $\vec{x}[k..l]$. If $k > l$ then this vector is by convention the unique vector of dimension 0. Furthermore, $\leq_{\text{lex}}$ $(<_{\text{lex}})$ denote the (strict) lexicographic ordering on vectors, and $\vec{x}^\top$ denotes the transpose of $\vec{x}$.

Scalar and matrix product are denoted by juxtaposition. Element (i, j) of matrix A is denoted by $A_{i,j}$. $\text{rank}(A)$ denotes the row rank of A. $A|_{i,\cdots,j}$ is the matrix that is composed of rows i to j of matrix A.

Definition 1. A *polyhedron* is the finite intersection of halfspaces. A *polytope* is a bounded polyhedron.

A $\mathbb{Z}$-*polyhedron* (a $\mathbb{Z}$-*polytope*) is the intersection of a polyhedron (polytope) and a lattice.

If not stated otherwise we mean $\mathbb{Z}$-polyhedra ($\mathbb{Z}$-polytopes) when we speak of polyhedra (polytopes).

2.3 Restrictions of the Input Program

As source language we use a subset of an imperative language like Pascal, Modula, C or Fortran. The syntax used in this thesis is self-explanatory, and we expect the reader to be familiar with the basic concepts of imperative languages. Thus, we focus immediately on the *restrictions* which we impose on general imperative programs:

- The only *data structures* considered are arrays. Extensions to records (structures) or unions (variant records) are straight-forward (but are not treated in this thesis), whereas aliasing mechanisms or pointers cannot be integrated easily.

- The only *control structures* are for loops and while loops. Conditionals can be modeled by while loops with at most one execution of the loop body; they are not treated explicitly in this thesis. Procedure and function calls can be integrated by considering them as a simultaneous assignment to those actual parameters which can be modified by the call, e.g., all reference parameters.

For technical reasons we add another constraint:

- In order to make dataflow analysis efficient or even feasible, the array indices must be affine functions in loop indices of surrounding loops and in structure parameters.

Please note that we inherit all these restrictions from the basic polytope model—they are not limitations due to the presence of while loops.

Note further that the basic polytope model also has the limitation that all occuring loops must be affine loops; the elimination of this restriction is the main contribution of this thesis.

2.4 Basic Model, Extensions and Parallelization

This section briefly presents the general technique of parallelization in the polytope model and proposes the basic idea of how to integrate while loops. A more detailed description of each parallelization step is deferred to the next chapter.

2.4.1 Parallelization of for Loops in the Polytope Model

Idea. The polytope model represents the atomic iteration steps of d perfectly nested for loops as the points of a polytope in $\mathbb{Z}^d$; each loop defines the extent of the polytope in one dimension. The faces of the polytope correspond to the bounds of the loops; they are all known at compile time. This enables the discovery of maximal parallelism (relative to the choices available within the model) at compile time.

Technique. The parallelization in the polytope model, described in [40], proceeds as follows (Figure 2.1, graphical representation for $n=3$).

First, one represents d perfectly nested source loops into a d-dimensional polytope where each loop defines the extent of the polytope in one dimension. We call this polytope the *index space* and denote it by $\mathcal{I}$ ($\mathcal{I} \subset \mathbb{Z}^d$). Each point of $\mathcal{I}$ represents one iteration step of the loop nest. The coordinates of the point are given by the values of the loop indices at that step; the vector of these coordinates is called the *index vector*.

Next, one applies an affine coordinate transformation $\mathcal{T}$, the *space-time mapping*, to the polytope and obtains another polytope in which some dimensions lie exclusively in space and the others lie exclusively in time. In other words, the new coordinates represent explicitly the (virtual) processor location and the time of execution of every computation of the target program. In Figure 2.1 the space-time mapping is given by $p = j$, $t = i+j$. We call the transformed polytope the *target space* and denote it by $\mathcal{TI}$.

```
for i := 0 to n do                      for t := 0 to 2n + 2 do
  for j := 0 to i + 2 do                  parfor p := max(0, t−n) to min(t, ⌊t/2⌋ + 1) do
    A(i, j) := A(i − 1, j)                  A(t−p, p) := A(t−p−1, p)
               +A(i, j − 1)                              +A(t−p, p−1)
  enddo                                   enddo
enddo                                   enddo
```

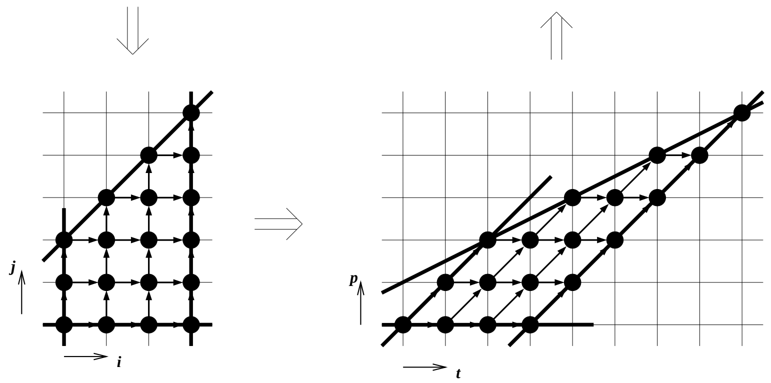

Figure 2.1: Parallelization in the model

Finally, one translates this polytope back to a nest of target loops, where each space dimension corresponds to a parallel loop and each time dimension corresponds to a sequential loop.

By-statement mapping. The model described up to this point can only handle perfectly nested loops. This severe restriction can be relaxed by applying all techniques mentioned so far to every statement in the body separately, instead of applying it to the body of as a whole. In this extension every statement gets its own index vector, its own source and target polytope and its own space-time mapping [19, 37]. Thus, the symbols denoting the polytopes and the transformation are indexed with the name of the statement. An *operation* of the program is identified by the pair consisting of the name S of a statement and its index vector $\vec{i}$; we write this $\langle S, \vec{i} \rangle$. The set of all operations is denoted by Ω.

Of course the introduction of by-statement space-time mappings complicates the generation of target code considerably; possible solutions are given in [12, 17, 37, 61].

We use the statement-based extension of the model. If we do not specify a specific statement explicitly, we mean any statement.

2.4.2 Parallelization of while Loops in the Polyhedron Model

A while loop is commonly denoted by while *condition* do *body*; in contrast to for loops there is no explicit loop index. However, since the polytope model is based on such indices, we must add loop indices to while loops. Therefore, we prefer a while loop notation as in the programming language PL/1 which contains an explicit index:

$$\textsf{for}\ index := lb\ \textsf{while}\ condition\ \textsf{do}\ body$$

where the lower bound lb is an affine expression in outer loop indices and structure parameters.

while loops without an explicit index can simply be given one with an arbitrary name and with an arbitrary affine expression lb as lower bound; usually lb is zero in the source program but, in general, it is not zero in the target program. The index value is incremented automatically after each iteration (as in for loops).

After adapting the notation of while loops to the model, we now discuss the consequences for the model. The extent of the index space of a statement in any dimension is given by the number of iterations of the loop spawning this dimension (Section 2.4.1). However, the upper bound of a while loop is unknown at compile time. Therefore, the index space is unbounded at compile time and, thus, not a polytope but a polyhedron. That is the reason why we call our extended model the *polyhedron model*.

At run time, a nest with while loops executes only a subset of the infinite index space $\mathcal{I}$. We call this subset (which can, in general, not be predicted at compile time) the *execution space* and name it $\mathcal{X}$. Note that $\mathcal{X}$ need not be convex, and thus need not be a polytope. This property poses one of the central problems concerning the generation of target programs. We shall see that the same difficulties also occur for non-affine for loops; more details are given in Section 4.2 and an appropriate solution is presented in Chapter 5.

For consistency reasons the non-convex set of points enumerated by non-affine for loops is also called the *execution space* and named $\mathcal{X}$. The index space of non-affine for loops is the convex—possibly also infinite—approximation which results from omitting all non-convex bounds. Thus, index spaces are always convex.

Remark. Note that we assume that the source program terminates.

Example 1. Consider the loop nest in Figure 2.2.

```
w1:  for i := 0 while cond1(i) do
w2:      for j := 0 while cond2(i, j) do
S:           body(i, j)
         enddo
     enddo
```

Figure 2.2: Two nested while loops

Figure 2.3 shows the index space (a) and a possible execution space (b) of statement S.

Remark. The termination detection of while loops requires some computations at run time. These computations must be treated as regular statements, i.e., they must have, for example, their own index and execution spaces. We call these statements *loop statements*.

Since loop statements are treated as regular statements, the dimensionality of their index space should be equal to the *depth* of the loop statement, i.e., the number of surrounding loops of the statements—as for the statements of the loop body. But this does not make sense for loop statements whose computed values vary per iteration, as is the case for loop

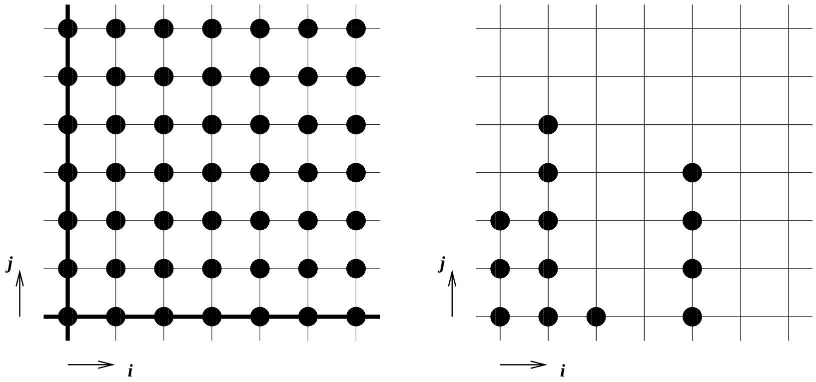

Figure 2.3: (a) Index space (b) Possible execution space

statements representing while loops. In this case, the dimensionality of the index space of the loop statement is the depth plus 1.

Remark. We assume that the for loop bounds are evaluated once before the execution of the loop as in Fortran, Pascal and Modula, not before every iteration as in C. (In fact, C for loops are disguised while loops.) Thus, the dimensionality of the index space of the loop statement of a for loop is equal to the depth of this loop statement.

Remark. Since loop statements guard the execution of the statements in the loop body, we usually overlay the execution spaces of the loop statements and the statements in the loop body in graphical representations. In such an *overlay representation* black dots represent the computation points of the loop body, whereas dots in the various shades of gray represent the *testing points* of loop statements.

In our graphical representations, the priorities of the axes are horizontal over vertical over depth, if priorities are considered at all. I.e., the horizontal axes is enumerated by the outer loop, and the other axes follow outside-in according to their priority.

Example 2. Figure 2.4 shows the construction of the execution space of statement S of Example 1 in overlay representation: (a) to (c) each depicts one possible execution space for the statements w_1, w_2 and S, respectively. (d) shows the overlay of (a) to (c), where lighter points are obscured by darker points. Consequently, the only visible points are the computation points of S and those testing points whose corresponding condition evaluates to *ff*.

2.5 An Example Application

Throughout this thesis we illustrate all parallelization steps by applying them to an algorithm for calculating the reflexive transitive closure of a finite, directed, acyclic, sparse graph which is given by its adjacency list. More formally, a graph is represented by a set *node* of nodes and, for every node, by the number *nrsuc* of its successors and the set *suc* of successor nodes. *rt* of n is the adjacency list of node n in the reflexive transitive closure.

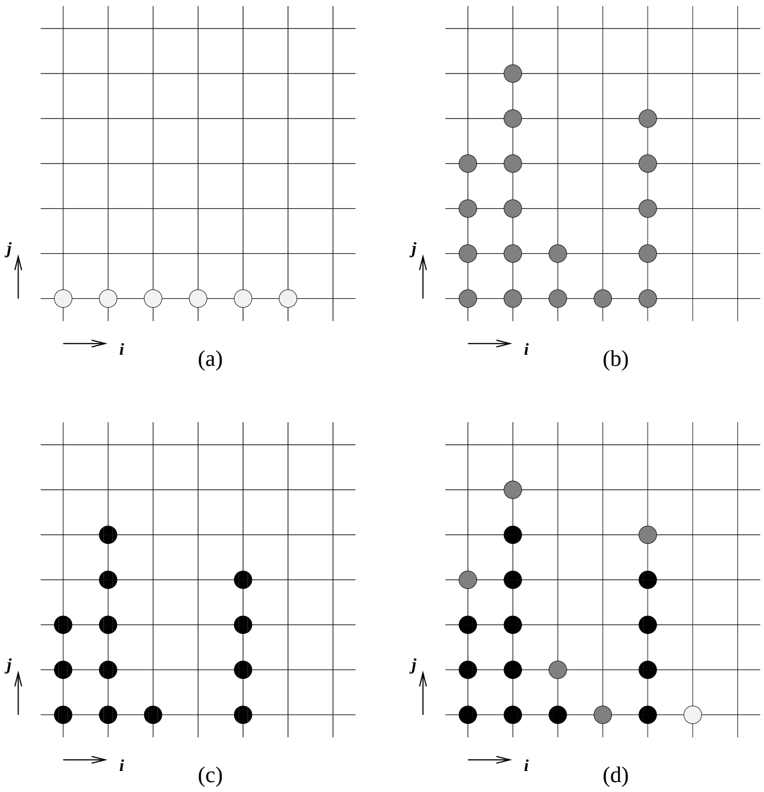

Figure 2.4: Execution space in overlay representation

Example 3. The graphs in Figure 2.5 are represented in the source program as follows:

n	*node*	*nrsuc*	*suc*	*rt*
0	A	0		A
1	B	1	C	B, C, A, E, D
2	C	2	A, E	C, A, E, D
3	D	0		D
4	E	1	D	E, D

The following source algorithm computes the reflexive transitive closure, under the assumption that the resulting adjacency lists rt are initially empty:

for every node n do
  add n to rt of n
  while there is a node m not yet considered in rt of n do
    for every successor ms of m do
      add ms to rt of n

Note that this algorithm may produce adjacency lists which contain multiple occurrences of some nodes. This is a suboptimal representation, but enforcing lists with unique elements spoils the parallelism; more on that later.

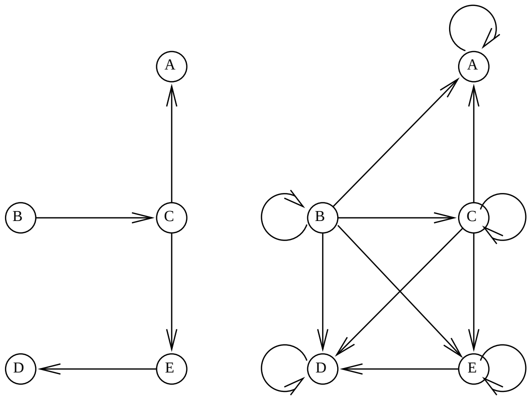

Figure 2.5: A graph and its reflexive transitive closure

Since the polyhedron model offers no methods for dealing with sets or lists (not yet, anyway) but excels on arrays, we use arrays in our concrete representation. *node* and *nrsuc* are one-dimensional arrays, *suc* and *rt* are two-dimensional. For the computation of the reflexive transitive closure we need an auxiliary one-dimensional array *nxt* which, for every node n, provides a pointer to the next free entry in the list of n's successors in the reflexive transitive closure. Initially all undefined array elements contain the value $\perp$; *rt* and *nxt* are undefined everywhere; *tag* must be initialized with *ff*. The purpose of *tag* is to mark nodes which have been visited so as to guarantee termination in graphs containing cycles. The domain of array *node* exceeds the number of nodes by 1 in order to accommodate the undefined element which forces termination of the outer while loop; the domain of arrays $rt[n]$ is unknown at compile time for every node n. The source program is given in Figure 2.6.

```
S1:  for n := 0 while node[n] ≠ ⊥ do
S2:    rt[n,0] := n
S3:    nxt[n] := 1
S4:    for d := 0 while rt[n,d] ≠ ⊥ do
S5:      if ¬tag[n, rt[n,d]] then
S6:        tag[n, rt[n,d]] := tt
S7:        for s := 0 to nrsuc[rt[n,d]] − 1 do
S8:          rt[n, nxt[n]+s] := suc[rt[n,d], s]
           enddo
S9:        nxt[n] := nxt[n] + nrsuc[rt[n,d]]
         endif
       enddo
     enddo
```

Figure 2.6: The source program

Note that some array indices are non-affine expressions in outer loop indices and parameters. This requires manual interaction for generating suitable input to dependence analysis tools and leads to an overly conservative estimation of the existing dependences (Section 3.1.5).

Let us now illustrate the index and possible execution spaces of this example.

The index space of statements S_2 and S_3 is $\{n \mid n \geq 0\}$, for statements S_5, S_6 and S_9 it is $\{(n,d) \mid n,d \geq 0\}$, and for statement S_8 it is $\{(n,d,s) \mid n,d,s \geq 0\}$.

The index spaces of statements S_1, S_4 and S_7 are $\{n \mid n \geq 0\}$, $\{(n,d) \mid n,d \geq 0\}$ and $\{(n,d) \mid n,d \geq 0\}$, respectively. (Remember that the dimensionality of index spaces of for loops is equal to the depth of the loop statement.)

For an illustration of possible execution spaces of statements S_1, S_4 and S_9 we refer to Figure 2.4 again: (a), (b) and (c) represent the execution spaces of statement S_1, S_4 and S_9, respectively, where index n corresponds to i and d corresponds to j.

We have proposed a way of integrating while loops into our computational model. In the following chapters we focus on the individual steps of the loop parallelization methods of the polytope model and present all necessary extensions to these methods for an extension to the polyhedron model.

Chapter 3

Important Parallelization Phases

This chapter describes the most important phases of the parallelization in the polytope model and the necessary extensions for the polyhedron model.

3.1 Dependence Analysis

In our approach, all limitations of parallelism are specified as *dependences*. Dependent operations must be executed in a predefined order, whereas independent operations may be executed in parallel. The following sections show that there are various kinds of dependences. All these kinds of dependences must be represented in a common dependence model which fits our computational model. This dependence model is the dependence graph defined in Section 3.1.4.

3.1.1 Data Dependence Analysis in the Polytope Model

Data dependence provides information about the flow of data. In imperative languages, data dependences boil down to conflicting accesses to memory cells. Bernstein expressed this already in 1966 in his famous conditions for the existence of dependences [7], which can be summarized as follows: two operations can only be data dependent if both access the same memory cell and at least one of the two accesses is a write access.

Unfortunately, data dependence analysis is only well developed for scalar variables and for arrays whose indices are affine functions in structure parameters and surrounding loop indices [3, 5, 47].

For a definition of data dependences in the case of scalars and arrays, we first need a refinement of the lexicographic order on operations.

Definition 2 (Sequential execution order $\prec$). For two operations $o_1 = \langle S_1, \vec{i_1} \rangle$ and $o_2 = \langle S_2, \vec{i_2} \rangle$

$$o_1 \prec o_2 \;\Leftrightarrow\; \vec{i_1}[1..k] <_{\text{lex}} \vec{i_2}[1..k] \;\vee\; (\vec{i_1}[1..k] = \vec{i_2}[1..k] \;\wedge\; S_1 \text{ is textually before } S_2),$$

where k is the number of loops surrounding both S_1 and S_2.

Definition 3 (Data dependence). An operation o_2 is *data dependent* on an operation o_1, written $o_1 \delta o_2$, if

- o_1 and o_2 refer to the same scalar or array, and, in the latter case, all indices of the array are identical,
- $o_1 \prec o_2$, and
- at least one of the two references is a write access.

o_1 is called the *source* and o_2 the *sink* of the dependence. A data dependence is called a *true dependence*, *anti dependence* or *output dependence* if only the reference in o_1, only the reference in o_2 or both references are write accesses, respectively. The three kinds of dependences are denoted by $\delta^t, \delta^a, \delta^o$, respectively.

If *spurious dependences* shall be avoided, one more restriction must be added:

- There is no operation o_3 such that $o_1 \prec o_3 \prec o_2$ which writes to the same scalar or array cell.

We call a true dependence which satisfies this additional constraint a *flow dependence* and denote it by δ^f.

In nests of affine loops this additional restriction enables us to determine, for every operation reading some variable, the precise operation that wrote to that variable most recently. With this information one can convert the source program to *single-assignment form*, in which all variables are replaced by sufficiently large array variables such that no array cell is written more than once.

Thus, this technique of *single-assignment conversion* avoids anti and output dependences as well as spurious dependences. Therefore, programs in single-assignment form usually have more parallelism—at the price of an increase in memory. There are algorithms for computing flow dependences and for single-assignment conversion in the case of nests of affine loops, e.g., [28].

Let us now define some additional technical concepts of dependence analysis. Let $\vec{i_1}$ and $\vec{i_2}$ be the index vectors of two dependent operations o_1 and o_2, respectively, reduced to common loop indices. Then, the difference $\vec{i_2} - \vec{i_1}$ is called a *dependence vector*. If the dependence vector is the zero vector the dependence is called *loop-independent*, otherwise it is called *loop-carried*.

Instead of enumerating every dependence separately, one often tries to use a common representation which subsumes all dependences caused by the same conflicting accesses. There are special cases in which this can be done easily: if all dependence vectors are identical we speak of a *uniform dependence*—in this case the common dependence vector is also called the *distance vector*; if the dependence vectors are affine functions in the index vectors, we speak of an *affine dependence* [3, 4, 52]. For affine dependences one sometimes abstracts from the precise affine function but uses what is called *direction vectors* instead. A direction vector is similar to a distance vector but it carries less information: $*$ is a wildcard for any arbitrary value and $+$ for any positive value, and juxtaposition denotes disjunction [63]. E.g., the direction vector $(0+, *)$ specifies dependences with dependence vectors $(0, \lambda)$ or (μ, λ) with $\mu, \lambda \in \mathbb{Z}$ and $\mu > 0$.

3.1.2 Data Dependence Analysis in the Polyhedron Model

Feautrier's method for data dependence analysis in the polytope model [28] has been adapted to loop nests containing while loops by Collard, Barthou and Feautrier [16]. In a loop nest with while loops one can, in general, no longer find the precise source of a dependence, but only a *set of possible* sources. This also has consequences for single-assignment conversion [11].

We use the techniques of Feautrier and Collard to compute the data dependences but we do not explore the issue of single-assignment conversion.

3.1.3 Control Dependences

Definition 4 (Control dependence). An operation o_2 is *control dependent* on an operation o_1, written $o_1 \delta^c o_2$, if whether o_2 is executed or not is determined by o_1.

Example 4. In the following program

S_1: if *condition* then
S_2: *body*
endif

S_2 is control dependent on S_1.

Like conditional statements, while loops introduce control dependences: every operation in the body of a while loop is control dependent on the computation of the while loop's termination condition at its own index vector.

In principle, this dependence is also present in affine loops but, since the loop bounds are known at compile time, all information necessary for a parallelization can be obtained without making these dependences explicit. In this case the loop statement itself is usually not considered in the parallelization: it is given neither a polytope nor a space-time mapping.

In addition to the control dependences just described, while loops have loop-carried dependences: the loop statement itself, i.e., the calculation of the termination condition, is control dependent on its predecessor. This is due to the fact that a while loop terminates as soon as its condition evaluates once to *ff*, and it does not restart whatever the values of the termination condition at the succeeding points are. We also call these control dependences while *dependences*.

The graphical representation of the while dependences in the overlay of the execution spaces of some nested while loops has the shape of a (possibly multi-dimensional) comb. Therefore, we also call the execution space an *execution comb* and refer to the iterations of one while loop with fixed outer loop indices as a *tooth* of the execution comb. Figure 3.1 depicts the execution spaces of statement S_9 and its surrounding loop statements in overlay representation.

3.1.4 Dependence Graph

The (full) *dependence graph* of a loop nest is the directed acyclic graph (Ω, E) whose vertex set Ω is the set of all operations of the loop nest and whose edge set E contains all dependences between the operations represented by the vertices. The dependence graph w.r.t. the index set

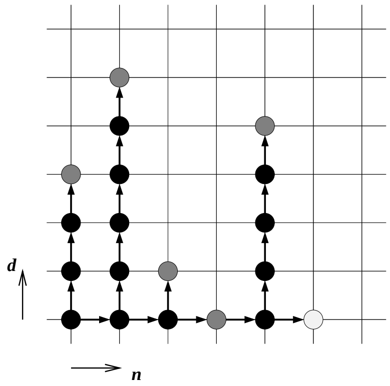

Figure 3.1: A possible execution space with control dependences

may be infinite, whereas the dependence graph w.r.t. the execution set is finite (but unknown at compile time).

Alternatively, some parallelization techniques work on the *reduced dependence graph* which is obtained from the full dependence graph by projecting all operations of one statement on a single node [21, 23]. This graph is always finite since it has one node per statement; on the other hand, it carries less information than the full dependence graph. To keep as much information as possible, every edge of the reduced dependence graph is usually labeled by the distance vector or the direction vector.

3.1.5 The Example

Control dependences. Based on the explanations in Section 3.1.3 we can list all control dependences of the program in Figure 2.6 on page 16. In Table 3.1 column *dist* specifies the distance vector of the dependences. For all three (non-affine) loops we have specified the zero distance vectors, meaning that the loop body's execution depends on the result of the computations in the loop bounds. We have also specified the while dependences for the two while loops (c_{10} and c_{16}). The dependences c_{18} to c_{21} represent the control dependences caused by the if clause.

Data Dependences. A parallelization requires first a data dependence analysis. For this purpose we use the tool Tiny [63], which takes as input a program and yields as output the direction vectors of all dependences in the program. With the help of this tool, we have obtained the dependence information in Table 3.2 (semi-automatically), where column *var* contains the name of the array which causes the dependence. The entries of column *dir* are the direction vectors.

Let us have a closer look at some dependences. In general, it is undecidable at compile time whether $A[B[\vec{i}]]$ is the same variable as $A[\vec{j}]$ if nothing is known about $B[\vec{i}]$. Therefore Tiny assumes that every access to an indirectly indexed array conflicts with every other access to the same array, e.g., $rt[n, nxt[n]{+}s]$ conflicts with every $rt[n, d]$. But we know the following program-specific properties.

Lemma 5. *In the sequential execution, the loop on d has the following invariant: $nxt[n]$ is the index to the first undefined element in $rt[n]$.*

nr	*type*	*from*	*to*	*dist*	*nr*	*type*	*from*	*to*	*dist*
c_1	ctrl	S_1	S_2	(0)	c_{12}	ctrl	S_4	S_6	$(0,0)$
c_2	ctrl	S_1	S_3	(0)	c_{13}	ctrl	S_4	S_7	$(0,0)$
c_3	ctrl	S_1	S_4	(0)	c_{14}	ctrl	S_4	S_8	$(0,0)$
c_4	ctrl	S_1	S_5	(0)	c_{15}	ctrl	S_4	S_9	$(0,0)$
c_5	ctrl	S_1	S_6	(0)	c_{16}	ctrl	S_4	S_4	$(0,1)$
c_6	ctrl	S_1	S_6	(0)	c_{17}	ctrl	S_7	S_8	$(0,0,0)$
c_7	ctrl	S_1	S_7	(0)	c_{18}	ctrl	S_5	S_6	$(0,0)$
c_8	ctrl	S_1	S_8	(0)	c_{19}	ctrl	S_5	S_7	$(0,0)$
c_9	ctrl	S_1	S_9	(0)	c_{20}	ctrl	S_5	S_8	$(0,0)$
c_{10}	ctrl	S_1	S_1	(1)	c_{21}	ctrl	S_5	S_9	$(0,0)$
c_{11}	ctrl	S_4	S_5	$(0,0)$					

Table 3.1: The control dependences

Proof. Induction on the loop index d:

Induction Base: When $d = 0$, the only defined values are $rt[n,0]$, for $n \geq 0$, and $nxt[n]$ is initialized to 1 for $n \geq 0$. Thus, the postulate holds at the beginning of the first iteration.

Induction Step: At each iteration of the loop on d, $nxt[n]$ is increased by the number of new values appended to positions $nxt[n]+s$. Thus, at the end of the iteration, $nxt[n]$ points again to the first undefined element.

Lemma 6. *Another invariant of loop d, for any n, is: $0 \leq d < nxt[n]$.*

Proof. The while condition holds at every step of the while loop on d, thus $rt[n,d] \neq \perp$. Therefore, with Lemma 5, $0 \leq d < nxt[n]$.

As a consequence, memory accesses of $rt[n, nxt[n]+s]$ and $rt[n,d]$ in the same iteration always refer to different array elements. Thus, we may drop any dependence which is caused by the update of $rt[n, nxt[n]+s]$ in statement S_8 and any read access to $rt[n,d]$ in the same iteration, i.e., with a direction vector with leading coordinates $(0,0)$, which applies to the dependences d_{18} and d_{25}. For the same reason, the direction vectors $(0,0+)$ of dependences d_{11}, d_{13}, d_{14}, d_{17}, and d_{27} can be changed to $(0,+)$.

Note that this optimization is not necessary—neither for finding parallelism, nor for illustrating the concepts we are going to introduce. However, it thins the dependence graph out enough to permit a one-dimensional schedule (Section 3.2.3). Without it, the best schedule derivable with present techniques of array dependence analysis has two dimensions [29, 30]. It is to be hoped that methods of set dependence analysis, yet to be developed, will make such manual, problem dependent adjustments obsolete.

The fact, pointed out earlier, that the algorithm does not produce an optimal representation —the adjacency lists may contain multiple entries—is essential in making the optimization work. If we extracted these multiple entries, the number of added nodes in the loop on s could drop below the increment of $nxt[n]$ in statement S_7, which would foil the induction step in the proof of Lemma 5.

nr	*type*	*from*	*to*	*var*	*dir*	*nr*	*type*	*from*	*to*	*var*	*dir*
d_1	flow	S_2	S_4	*rt*	(0)	d_{18}	anti	S_8	S_8	*rt*	$(0,0,+)$
d_2	flow	S_2	S_5	*rt*	(0)	d_{19}	anti	S_8	S_8	*rt*	$(0,+,*)$
d_3	flow	S_2	S_6	*rt*	(0)	d_{20}	anti	S_8	S_9	*nxt*	$(0,0+)$
d_4	flow	S_2	S_7	*rt*	(0)	d_{21}	flow	S_8	S_4	*rt*	$(0,+)$
d_5	flow	S_2	S_8	*rt*	(0)	d_{22}	flow	S_8	S_5	*rt*	$(0,+)$
d_6	flow	S_2	S_9	*rt*	(0)	d_{23}	flow	S_8	S_6	*rt*	$(0,+)$
d_7	output	S_2	S_8	*rt*	(0)	d_{24}	flow	S_8	S_7	*rt*	$(0,+)$
d_8	flow	S_3	S_8	*nxt*	(0)	d_{25}	flow	S_8	S_8	*rt*	$(0,0,+)$
d_9	flow	S_3	S_9	*nxt*	(0)	d_{26}	flow	S_8	S_8	*rt*	$(0,+,*)$
d_{10}	output	S_3	S_9	*nxt*	(0)	d_{27}	flow	S_8	S_9	*rt*	$(0,0+)$
d_{11}	anti	S_4	S_8	*rt*	$(0,0+)$	d_{28}	output	S_8	S_8	*rt*	$(0,+,0)$
d_{12}	anti	S_5	S_6	*tag*	$(0,0+)$	d_{29}	anti	S_9	S_9	*nxt*	$(0,+)$
d_{13}	anti	S_5	S_8	*rt*	$(0,0+)$	d_{30}	anti	S_9	S_8	*rt*	$(0,+)$
d_{14}	anti	S_6	S_8	*rt*	$(0,0+)$	d_{31}	flow	S_9	S_8	*nxt*	$(0,+)$
d_{15}	flow	S_6	S_5	*tag*	$(0,+)$	d_{32}	flow	S_9	S_9	*nxt*	$(0,+)$
d_{16}	output	S_6	S_6	*tag*	$(0,+)$	d_{33}	output	S_9	S_9	*nxt*	$(0,+)$
d_{17}	anti	S_7	S_8	*rt*	$(0,0+)$						

Table 3.2: The data dependences

3.2 Schedule and Allocation

3.2.1 Space-Time Mapping in the Polytope Model

The problem of scheduling computations (in time) and allocating them (in space) has received a lot of attention in the framework of polytopes, from the seminal work of thirty years ago by Karp, Miller and Winograd [36] to many recent extensions [10, 29, 30, 51, 52].

Definition 7 (Schedule, allocation, space-time matrix). Let Ω be a set of operations, (Ω, E) their dependence graph, and r, r' integer values.

- Function $t : \Omega \to \mathbb{Z}^r$ is called a *schedule* if it preserves the data dependences:

  $$(\forall\ x, x' : x, x' \in \Omega \wedge (x, x') \in E : t(x) <_{\text{lex}} t(x'))$$

  The schedule that maps every $x \in \Omega$ to the *first possible* time step allowed by the dependences is called the *free schedule*.

- Any function $a : \Omega \to \mathbb{Z}^{r'}$ can be interpreted as an *allocation*.

Most parallelization methods based on the polytope model require the schedule and allocation to be affine functions for every statement S:

$$\begin{array}{l}(\exists\ \lambda_S, \alpha_S\ : \lambda_S \in \mathbb{Z}^{r\times d}\ \wedge\ \alpha_S \in \mathbb{Z}^r\ : (\forall\ i : i \in \mathcal{I}_S : t(\langle S, i\rangle)\ =\ \lambda_S\, i + \alpha_S)) \\ (\exists\ \sigma_S, \beta_S\ : \sigma_S \in \mathbb{Z}^{r'\times d}\ \wedge\ \beta_S \in \mathbb{Z}^{r'}\ : (\forall\ i : i \in \mathcal{I}_S : a(\langle S, i\rangle)\ =\ \sigma_S\, i + \beta_S))\end{array}$$

The matrix $\mathcal{T}_S$ formed by λ_S and σ_S is called a *transformation matrix* or *space-time matrix*:

$$\mathcal{T}_S\ =\ \begin{pmatrix} \lambda_S \\ \sigma_S \end{pmatrix}$$

We call the images $\mathcal{T}_S(\mathcal{I}_S)$ and $\mathcal{T}_S(\mathcal{X}_S)$ of the index and the execution space of a statement S the *target polyhedron* or *target index space* and the *target execution space* and denote them by $\mathcal{T}\mathcal{I}_S$ and $\mathcal{T}\mathcal{X}_S$, respectively.

Recently, a relaxation to piecewise affine functions for schedule and allocation has been investigated [10, 29, 30, 51, 52].

For technical reasons we require at some points the invertibility of the space-time matrix $\mathcal{T}$. If $\mathcal{T}$ is not invertible, one proceeds in three steps: first, one constructs an auxiliary transformation matrix $\overline{\mathcal{T}}$ from $\mathcal{T}$ by eliminating linearly dependent rows and, if necessary, adding new, linearly independent rows to get an invertible square matrix, second, one uses $\overline{\mathcal{T}}$ as the transformation matrix, and, third, one re-inserts the eliminated rows [61]. The rows added in the first step can be viewed as a refinement of the time computed by the scheduler. (Note that laying out these added dimensions in space would also be correct, but this might violate some locality which is intended by the allocator; interpreting these additional dimensions as refined time hampers neither schedule nor allocation.)

This technique allows us to assume—without loss of generality—that all space-time matrices are invertible. When necessary, we shall refer to $\overline{\mathcal{T}}$ as the *essential transformation matrix*.

Note that the re-insertion of linearly dependent rows in the third step can lead to transformation matrices which have more rows than columns, i.e., the target space can have more dimensions than the source space. The dimensionality of the image of the source space, however, is the same as the dimensionality of the source space since the essential part of the transformation comes from the invertible $\overline{\mathcal{T}}$—this image is only embedded in a higher-dimensional space.

There are many algorithms for computing a schedule or an allocation, not only in the case of uniform dependences [36, 39, 50, 54] but also in the case of affine dependences [20, 22, 29, 30].

We usually use the scheduler of Darte/Vivien [20] which works on the reduced dependence graph. The quality of the generated schedule falls a bit behind that of Feautrier's method [29, 30], but the computation of the schedule is much faster.

For finding allocations we apply Feautrier's method [31], which is based on the owner computes rule and tries to minimize communications with a greedy heuristics.

3.2.2 Space-Time Mapping in the Polyhedron Model

The extension of existing space-time mapping methods from affine loop nests to loop nests containing `while` loops has been worked out by Collard [13]. In principle, the scheduling methods of the polytope model are suitable for `while` loops without any change; the only addition necessary is a mechanism for handling the imprecise output of the dataflow analysis.

3.2.3 The Example

When we apply the scheduling methods of Darte/Vivien [20] and the allocation method of Feautrier [31] to our example program we obtain the schedules and allocations of Table 3.3. The "leak" in the schedule, i.e., the fact that the time steps $n+2$ and $n+3$ are missing, is due to the suboptimal scheduling method of Darte/Vivien; it would not occur in the optimal schedule.

Note that our implementation of Feautrier's allocator allows to vary the number of allocation dimensions—according to Definition 7 it can be chosen freely. Table 3.3 shows the one- and the three-dimensional allocation; the two-dimensional allocation is uninteresting since, in that case, the schedule is linearly dependent on the allocation of every statement.

statement	schedule	1-dim. allocation	3-dim. allocation
S_1	n	n	$(n,0,0)$
S_2	$n+1$	n	$(n,0,0)$
S_3	$n+1$	n	$(n,0,0)$
S_4	$n+4d+4$	n	$(n,d-1,0)$
S_5	$n+4d+5$	n	$(n,d-1,0)$
S_6	$n+4d+6$	n	$(n,d-1,0)$
S_7	$n+4d+6$	n	$(n,d-1,0)$
S_8	$n+4d+7$	n	(n,d,s)
S_9	$n+4d+8$	n	$(n,d,0)$

Table 3.3: The space-time mapping

Note that, in this example, the schedule and the allocation are linearly dependent. Therefore, as written above, the target space of, e.g., statement S_8 w.r.t. the three-dimensional allocation is four-dimensional, although the index space is only three-dimensional.

3.3 Generation of Target Programs

3.3.1 Generation of Target Loops in the Polytope Model

The result of a space-time mapping of a source polyhedron is again a polyhedron. Since the result of automatic parallelization ought to be a parallel program, not a geometrical object, we have to re-describe the target polyhedron by a nest of loops, where dimensions in time (enumerated by the schedule) become sequential loops and dimensions in space (enumerated by the allocation) become parallel loops. This process is called the *scanning* of the target space.

For this purpose, one first chooses the order of the loops. The target loop nest specifies *asynchronous parallelism* if the outer loops are the parallel ones, and *synchronous parallelism* if the outer loops are the sequential ones [40]; Banerjee calls this vertical and horizontal parallelism [5], respectively. Of course, a mixture of both variants is also possible.

Then, one computes loop bounds, such that a bound of an outer loop must not depend on the indices of inner loops. For this purpose, the inequality system describing the target polyhedron must be rewritten: for every dimension of the target loop nest we eliminate successively, inside out, all occurrences of inner loop variables in the inequality system. This method is known as *Fourier-Motzkin elimination*, was developed in about 1827, and is presented, for example, in [4], pp. 81–94. From the resulting description of the target space the target loop bounds can be read off immediately [1]. Several extensions to this simple method of computing target loops have been proposed, e.g., [9, 12, 37, 61]. They do not change the basic method but only extend its applicability.

When the space-time matrix $\mathcal{T}$ is not *unimodular*, i.e., when its inverse is not an integer matrix, $\mathcal{T}\mathcal{I}$ contains "holes", i.e., it is not convex even though $\mathcal{I}$ is [6]. More precisely, the lattice of $\mathcal{TI}$ is coarser than the lattice of $\mathcal{I}$. In this case, one has to take care that the target loops do not enumerate the holes. Luckily, non-unimodular mappings distribute holes evenly throughout the target space. Therefore, there is always a target loop nest that scans $\mathcal{TI}$ precisely—whether $\mathcal{T}$ is unimodular [1] or not [32, 65].

3.3.2 Extensions for the Most General Case of the Polytope Model

Since code generation for the polyhedron model is the focus of this work, we describe first the most general technique for code generation in the polytope model. S. Wetzel [61] presents a method for code generation which can be applied to non-unimodular, piecewise affine by-statement transformations of imperfectly nested loops where, in addition, the space-time matrices need neither be square nor of full rank. We exploit her results for the extension to code generation in the polyhedron model.

Section 3.2.1 describes how non-square or singular transformation matrices can be tackled. The basic observation of [61] is that all remaining extensions (piecewise affinity, by-statement mapping, imperfect loop nests) can be treated the same way.

As described previously, every statement, together with its enclosing loops, is considered individually. In addition, if the space-time mapping of a statement is piecewise, its index space is divided into the subspaces defined by the pieces, and the statement is copied and assigned to everyone of the resulting subspaces; every resulting pair of a subspace and its statement is called a *program part* and can be transformed individually, since it has its own affine (not piecewise!) mapping, which might be non-unimodular but will be of full rank. This method yields a set of target spaces, one per program part, which can be scanned individually with standard methods (e.g., [1, 65]).

The main task remaining is to combine all target program parts. For this purpose, Wetzel mainly offers two methods: merging at run time and merging at compile time.

The first method consists of finding a convex set S which encloses the union of all target program parts (e.g., the convex or rectangular hull). Then, the generated loop nest enumerates S, and the statement of every program part is guarded by a condition expressing the exact bounds of the target program part.

The second method consists of computing all intersections and overlaps of the target program parts and yields an imperfect target loop nest, which enumerates successively regions which contain the same set of overlapping program parts. This avoids conditional statements in the loop nest.

However, the disadvantage of the second method is, that, in the presence of symbolic constants, the intersections of the target program parts cannot be computed at compile time. Since the order of the structure parameters is not known, this method generates one target program for every possible order of the values of the bounds of the target program parts containing symbolic constants, thus leading to $O(n!)$ cases, where n is the number of symbolic constants.

In the presence of while loops, merging at compile time is impossible. Thus we exploit the first method.

Example 5. Let us convert all while loops in our example to for loops with affine bounds. The resulting program is senseless but it sets the stage for the code generation for the nest with while loops. The code, obtained by applying the methods of [61], is given in Figure 3.2.

```
S1: for n := 0 to N do
S2:     rt[n,0] := n
S3:     nxt[n] := 1
S4:     for d := 0 to D do
S5:         if ¬tag[n, rt[n,d]] then
S6:             tag[n, rt[n,d]] := tt
S7:             for s := 0 to S do
S8:                 rt[n, nxt[n]+s] := suc[rt[n,d], s]
                enddo
S9:             nxt[n] := nxt[n] + nrsuc[rt[n,d]]
            endif
        enddo
    enddo
```

Figure 3.2: A modified source program

Let us use the one-dimensional allocation and the schedule of Table 3.3. The asynchronous target program is given in Figure 3.3.

Note first that we drop the loop statements (S_1, S_4, and S_7), since these statements do not appear in the polytope model, but for simplicity we do not tighten the schedule.

It is easy to recognize that all statements are guarded by a condition. This is due to the fact that the program parts of the statements all have different offsets in the time dimension, but the loop in this dimension must enumerate all possible time steps—the guards ensure that every statement is only executed in its own target index space.

The modulo operations in the guards, denoted by %, are caused by the non-unimodularity of the transformation.

The source index space of statement S_8 has three dimensions, but the schedule and the allocation together only enumerate two dimensions. As described previously, we add a row (0 0 1) to the transformation matrix and view this additional dimension as a refinement of time. In [61], such loops only surround the relevant statements—the outermost loops only enumerate all necessary coordinates for the dimensions defined by schedule or allocation.

If every node has a local copy of the graph when our function is called, there is only one (non-local) communication for our allocation in the original example which comes from the unit control dependence at level 1. Since this dependence does not exist in the modified source program (there are no while loops), there is no need for communications or barrier synchronizations; all processors work independently.

3.3.3 Generation of Target Loops in the Polyhedron Model

This last phase of an automatic parallelization in the polytope model changes seriously if one allows non-affine loops. We are not aware of any work on this area before ours. According solutions to the arising problems are presented in the following chapters.

```
parfor p := 0 to N do
  for t_1 := p to max(p+1, p+4D+8) do
    if p+1 = t_1 then
      rt[p,0] := p
      nxt[p] := 1
    endif
    if (p+5) ≤ t_1 ≤ (p+4D+5) and
                 (t_1-p-5)%4 = 0 then
      if_cond[p, (t_1-p-5)/4] := not tag[p, rt[p, (t_1-p-5)/4]]
    endif
    if (p+6) ≤ t_1 ≤ (p+4D+6) and
                 (t_1-p-6)%4 = 0 and if_cond[p, (t_1-p-6)/4] then
      tag[p, rt[p, (t_1-p-6)/4]] := tt
    endif
    if (p+8) ≤ t_1 ≤ (p+4D+8) and
                 (t_1-p-8)%4 = 0 and if_cond[p, (t_1-p-8)/4] then
      nxt[p] := nxt[p] + nrsuc[rt[p, (t_1-p-8)/4]]
    endif
    if (p+7) ≤ t_1 ≤ (p+4D+7) and
               (t_1-p-7)%4 = 0 and if_cond[p, (t_1-p-7)/4] then
      for t_2 := 0 to S do
        rt[p, t_2+nxt[p]] := suc[rt[p, (t_1-p-7)/4], t_2]
      enddo
    endif
  enddo
enddo
```

Figure 3.3: Target code of the modified program

3.3.4 Re-indexation in the Loop Body

For completeness, let us mention the final step of a target code generation: the replacement of the source loop indices by target indices. The simplest solution is to apply the inverse of the space-time matrix [40, 61].

Simpler array indices (and thus a better performance) of the target program are achieved by the method of Collard [12], which completely rearranges the arrays. We do not dwell on this task any further, since it is independent of whether the source loops are while loops or for loops.

Chapter 4

Classification of Loops

Before we start on the technical details, let us give an overview of the variety of nested loops that can occur in imperative programs. Let us first state some basic properties.

4.1 Properties of Loops and Loop Nests

The following facts are either trivial (but worth stating explicitly) or can be found in any textbook on linear programming, e.g., [44, 55].

- The set of points enumerated by an affine loop nest is the intersection of a (convex) polytope and a lattice, i.e., a $\mathbb{Z}$-polytope.
- A $\mathbb{Z}$-polytope can be enumerated (scanned) by a loop nest whose bounds are affine expressions in outer loop indices and structure parameters [1].
- The image of a convex set under an affine transformation is a convex set.
- The image of a $\mathbb{Z}$-polytope ($\mathbb{Z}$-polyhedron) under an affine transformation of full rank is a $\mathbb{Z}$-polytope ($\mathbb{Z}$-polyhedron), perhaps with a different underlying lattice.
- The set of coordinates enumerated by any loop within a loop nest with fixed outer indices is the intersection of a one-dimensional convex set along the dimension spanned by the loop and a lattice, i.e., a one-dimensional $\mathbb{Z}$-polyhedron.
- Therefore, the set of points enumerated by a loop nest is the union of one-dimensional $\mathbb{Z}$-polyhedra.
- In general, the union of convex sets is not convex and the union of $\mathbb{Z}$-polyhedra is not a $\mathbb{Z}$-polyhedron.
- The set of points enumerated by a loop nest is the intersection of a (not necessarily convex) set of points and a lattice.
- In general, the points of the intersection of a non-convex set and a lattice cannot be scanned by a loop nest.

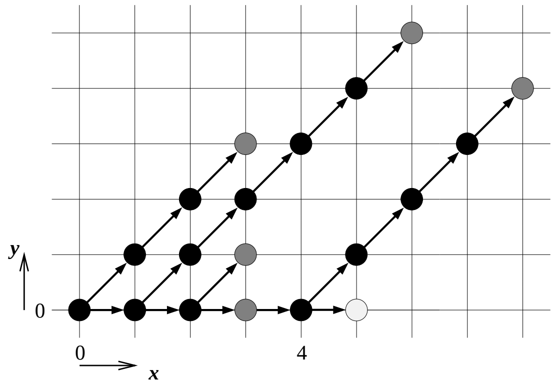

Figure 4.1: Unscannable target execution comb

These observations have a serious impact on the target code generation: a source loop nest may have a non-convex execution space, which cannot be enumerated by any loop nest after an affine transformation is applied.

Example 6. Let us apply the transformation

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} n \\ d \end{pmatrix}$$

to the execution comb in Figure 3.1 on page 21. The resulting target execution comb is presented in Figure 4.1. Let us consider, e.g., the line $x=4$. This line contains holes whose distribution depends on the upper bound of the inner while loop which, in turn, depends on the index of the outer while loop and is only known at run time. Thus, at compile time, we cannot generate a loop that enumerates precisely those points of the transformed execution comb which are located on the line $x=4$.

Of course, not all target execution spaces have this property. We call a set of points *scannable* iff there exists a loop nest which enumerates every point of the set once and no other point; otherwise the set is called *unscannable*.

A more detailed and formal treatment of scannability is given in Chapter 5. In the remainder of the current chapter we only need to be aware of the existence of such a problem.

4.2 Classification

Prevalently, only two types of loops are distinguished in the literature: for loops whose bounds are known at compile time and while loops whose iteration number, i.e., whose upper bound, is not known before run time. As we shall see, this distinction is not sufficient for a parallelization in the polyhedron model, esp. for target code generation.

Therefore, we propose a finer classification of loops and outline the impact of each class on the parallelization and the necessary code generation methods. The crucial factors in the classification are when the bounds of the loop can be determined and which form they take. As in the Chomsky hierarchy of formal languages, the larger the class, the lower the number we give it.

In effect, we classify loops individually and treat them individually according to their class. Note, however, that the class of a loop in a nest may depend on its outer loops.

We introduce five classes:

Class 4: Affine Loops. The bounds of these loops are affine expressions in the indices of the outer loops and in the structure parameters. These loops can be treated in the polytope model.

Example:

for $i := 0$ to n do
  for $j := 0$ to $i + 5$ do
    $\mathit{body}(i, j)$
  enddo
enddo

Class 3: Convex Loops. If the loop, together with the loops enclosing it, enumerates a convex set, of course intersected by a lattice (the source space), then there must be a loop nest which enumerates precisely the points of the set's image (the target space) under the space-time mapping, i.e., the target space is scannable. But there is no general mathematical framework (similar to Fourier-Motzkin elimination for Class 4) for identifying this loop nest.

The requirement that the check for convexity must be possible at compile time restricts the loop bounds to functions in the outer loop indices and structure parameters.

Example:

for $i := 0$ to n do
  for $j := 0$ to $\left\lceil \sqrt{i} \right\rceil$ do
    $\mathit{body}(i, j)$
  enddo
enddo

Note that there are a lot of extensions to non-linear analysis, e.g., [2, 43, 49], but they all focus on dependence analysis. The technique of [43] can (under some conditions) transform polynomial constraints to an (unbounded) set of piecewise linear constraints. This might sometimes allow to convert a loop of Class 3 to a loop of Class 4. However, we are not aware of any mathematical framework which can deal properly with loops of Class 3. Therefore, we treat loops of Class 3 as loops of Class 2 in this thesis.

Class 2: Arbitrary for Loops. The next larger class of loops contains loops whose number of iterations is not known at compile time, but is known when the execution of the loop begins. The bounds are arithmetic expressions in arbitrary variables and parameters. These loops are usually written as for loops, even though the bounds must be calculated at run time.

Example:

for $i := 0$ to n do
  for $j := 0$ to $A[i]$ do
    $\mathit{body}(i, j)$
  enddo
enddo,

for some array A.

Note that due to our semantics of for loops an occurrence of index j in the upper bound of the loop does not make sense, since the bound is evaluated only once.

If a loop of Class 2 is contained in a loop nest, then the image of the nest's index set is, in general, unscannable. Therefore, we must scan a superset of the image and prevent the points which are not in the image from execution. For this purpose, we consider control dependences with dependence vector $\vec{0}$ from the computation of the loop bound to all statements of the loop body. These dependences reflect that the maximal number of iterations can and must be calculated before the operations of the body are executed.

For Classes 3 and 4 such control dependences need not be considered since the transformed loop bounds capture all required information. However, if the space-time mapped bounds of convex loops cannot be computed precisely at compile time but only estimated, then enumerating a superset of the image and taking explicit care of the control dependences becomes necessary to exclude those points from execution which are not in the image.

Class 1: Static while Loops. In many while loops, the upper bound is also fixed when the while loop starts execution—however, it is not given explicitly as an arithmetic expression but as a while condition which does not hold in some iteration. Consequently, there is a while dependence, i.e., a control dependence from one iteration to the next iteration of the while loop. Obviously the target loop bounds must be computed at run time.

Example:

```
for i := 0 to n do
   for j := 0 while A[i,j] > 0 do
      body(i,j)
   enddo
enddo,
```

where array A is not modified in the body.

However, a loop of Class 1 has no dependence from the loop body to the variables in its termination condition. This can be exploited as follows.

We call a while loop *robust* if its termination condition can be evaluated at an index beyond the termination index, without leading to undesired side-effects. We call a robust while loop *strict* if its termination condition evaluates to *ff* for *all* iterations beyond the termination index.

If a static while loop is robust and strict, arbitrarily many while conditions can be evaluated simultaneously. Since this method ignores the while dependences, we may call it *speculative execution*. In fact, this is the *ideal* case for speculation.

We may also regard such a loop as an unfavorably denoted loop of Class 2. However, note that there is still no expression bounding the number of iterations of the loop. Thus, partitioning is necessary (cf. Section 6.2).

If a static while loop is only robust but not strict, one can again evaluate speculatively as many conditions in parallel as there are processors. Subsequently, one can, in logarithmic time, find the minimal index for which the termination condition evaluates to *tt*, if any, or enumerate the next block of conditions. This method finally yields the maximal index of the while loop, which can then be used as the upper bound of a for loop replacing the while loop. We do not exploit this option further since it falls outside our model.

Class 0: Dynamic while Loops. In the most general case of loops, the number of iterations may be changed by the iterations of the loop body. The difference to loops of Class 1 is a data dependence from a statement in the loop body to the while condition. This has no consequences for the code generation.

Example:

for $i := 0$ to n do
  for $j := 0$ while $A[i,j] > 0$ do
    $body(i,j)$
  enddo
enddo,

where array A is modified in the body.

In the literature, a popular way of parallelizing while loops (Classes 1 and 0) is to divide the loop body into a hopefully small "control" and a hopefully more complex "rest" part, then to execute the while loop with the statements of the control part only in order to obtain the extent of the while loop, and finally to spawn the same number of iterations by a—hopefully parallel—for loop containing the statements of the rest part in its body [64].

Note that, according to this method, a loop of Class 1 has the property that the control part only consists of the termination condition.

We claim that the space-time mapping approach unifies and generalizes other approaches to the parallelization of general while loops [59, 64], and that it yields the same pipelined solutions—or better ones, since the methods described before do not add any non-existent data dependences and provided one uses the best available by-statement scheduler [29, 30].

Of course, the suggested classification is not the only possible one. M. Geigl [33] describes a variety of parameters that influence the possibilities of code generation. Mainly he describes refinements of our classification, e.g., he presents cases in which code generation can do more than the approach presented here.

4.3 The Example

Let us classify the loops in our example program of transitive closure on page 14.

The outermost loop is a typical member of Class 1. If we had stored the number of nodes in some variable, we would get a loop of Class 3 since, together with its (non-existing) enclosing loops, the resulting for loop enumerates a convex set; if the number of nodes were a symbolic constant, it would even be a loop of Class 4. Target code enumerating the transformed index space precisely can be generated, since it is convex regardless of whether the outermost loop is a for or a while loop. However, if we convert this loop to Class 3 or Class 4, we can omit the unit and null control dependence vectors, which must be cited in loops of Class 1. This may result in a better schedule.

The loop on d is of Class 0 since list $rt[n]$, which determines its termination, becomes longer as execution proceeds.

The innermost loop is of Class 2 since its number of iterations is fixed when the loop starts, but is not known at compile time. On the other hand, the number of iterations of this loop differs for every instance, i.e., for every iteration vector (n,d), and it cannot be guaranteed at compile time that the set of all points (n,d,s) enumerated is convex, since this set depends on the input graph which is not known before run time. Therefore, the innermost loop is not of Class 3.

In the next three chapters we focus on the code generation for loops of Class 2, 1 and 0. To ensure readability, the theoretical sections concentrate on the perfectly nested case, or, more precise, on one statement together with its surrounding loops. The extension of these ideas to imperfectly nested loops does not introduce theoretical but only technical problems, solutions to which are discussed in [33]. However, we use the solutions of [33] in this thesis in order to treat our example program of Section 2.5.

Chapter 5

Scannability

As we have seen in Section 4.1, there are unscannable sets. In Section 5.1 we try to tackle this problem in more detail and treat it more formally. In Section 5.2 we try to obtain scannable target execution spaces "by construction", i.e., we distinguish a class of transformation matrices which guarantee scannable target spaces. Section 5.3 shows a way of dealing with unscannable sets.

5.1 Scannable Sets

We have seen that the target execution comb of Example 6 on page 30 is unscannable since the line $x=4$ contains holes whose distribution is only known at run time. Thus, in order to formalize the definition of a scannable set, we must formalize the definition of a hole.

As denoted in Section 4.1, the set of points enumerated by one loop at some level l inside a nest with fixed outer loop indices is a one-dimensional $\mathbb{Z}$-polyhedron, i.e., the intersection of a one-dimensional convex set and a grid. In other words, if the loop at level l enumerates two points $(x_1, \cdots, x_l)$ and $(x_1, \cdots, x_{l-1}, x'_l)$ with $x_l < x'_l$, then it also enumerates all intermediate points $(x_1, \cdots, x_{l-1}, x''_l)$ with $x_l < x''_l < x'_l$ on the grid. This leads to the formal definition of a hole.

Definition 8 (Hole). Let $S \subset \mathbb{Z}^d$ be a set of coordinate vectors on a grid with an implicit order $\lhd$ on the dimensions of the grid (the order in which the coordinates are written down). Then, a coordinate vector $(x_1, \cdots, x_d) \in \mathbb{Z}^d$ is a *hole w.r.t. level* r *and order* $\lhd$, for $1 \leq r \leq d$, iff

$$\begin{array}{l}(x_1, \cdots, x_d) \notin S \ \ \wedge \ \ (\exists\, (x_1, \cdots, \hat{x_r}, *, \cdots, *), (x_1, \cdots, \check{x_r}, *, \cdots, *) \ : \\ \quad (x_1, \cdots, \hat{x_r}, *, \cdots, *), (x_1, \cdots, \check{x_r}, *, \cdots, *) \in S \ : \check{x_r} < x_r < \hat{x_r}),\end{array}$$

where $*$ stands for an arbitrary value.

A coordinate vector $(x_1, \cdots, x_d) \in \mathbb{Z}^d$ is a *hole w.r.t. order* $\lhd$ iff it is a hole w.r.t. some dimension and w.r.t. order $\lhd$.

Now, we can formally define scannable sets.

Definition 9 (Scannable set). A set S is *scannable w.r.t. a predefined order* $\lhd$ on the dimensions iff S does not contain a hole w.r.t. order $\lhd$.

A set S is *scannable* if it is scannable w.r.t. some order $\lhd$.

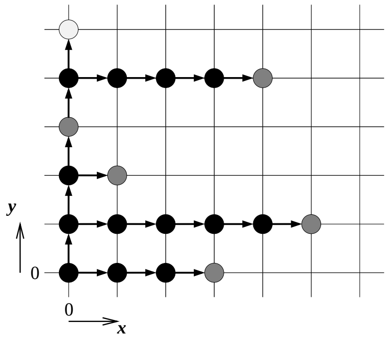

Figure 5.1: Unscannable comb w.r.t. the depicted order

For an illustration of these definitions we take another, very simple example and compare it with Example 6 on page 30.

Example 7. Let us again use the execution comb in Figure 3.1 on page 21 and apply the transformation

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} n \\ d \end{pmatrix}$$

to it. As in Example 6 on page 30 the line $x = 4$ contains holes whose distribution is only known at run time (Figure 5.1).

On the other hand, this transformation only represents loop interchange. Thus, if we scan first dimension y and then dimension x we can enumerate precisely all points—the source program does so! Therefore, the comb in Figure 5.1 is scannable, but unscannable w.r.t. the order in which x is the outer dimension, since, e.g., $(2, 2)$ is a hole w.r.t. level 2 for this order.

The target execution comb of Example 6 is unscannable, since the point $(4, 2)$ is a hole w.r.t. levels 1 and 2, regardless of the order of the dimensions.

Note that the scannability of transformed execution spaces is independent of which dimensions are in time and which are in space, or even, whether the transformation is a valid space-time mapping or not.

5.2 Scannable Transformations

After having introduced a formal definition of the sets which we can describe precisely (scannable sets), we now try to discover whether we can gain scannable target spaces "by construction". More precisely, we want to exploit the fact that all *source* programs enumerate sets of points, which therefore are scannable by definition. Thus, we are interested in identifying the class of *transformations* which preserves scannability. We also call such transformations *scannable*.

Note that the scannability of a transformation can never be a necessary condition for obtaining a scannable target execution space, since, e.g., convex source execution spaces lead to scannable target spaces for every transformation. Thus, we are only going to develop a sufficient condition for the scannability of a transformation.

Let us first introduce the following conventions:

- We refer to the loop immediately surrounding the statement at level l as *loop l*.

- The columns of the space-time matrix $\mathcal{T}$ are ordered (left to right) according to the (outside-in) order of the loops in the source loop nest.
- The rows of $\mathcal{T}$ are ordered (top to bottom) according to the (outside-in) order of the target loops which we want to generate. Which dimensions are in time and which are in space is immaterial.
- A column which corresponds to a loop of Class 3 or lower is called a *non-affine column*; the predicate *naff-col*(c) indicates whether column c is non-affine.

Remark. In the *polytope* model, only rows representing multi-dimensional time have a given order; the rows representing (virtual) space have no special positions, i.e., the choice of a synchronous or asynchronous target program does not influence the transformation matrix. However, in the *polyhedron* model, the order of the target loops is very important, as we shall see later on in this chapter. Therefore, we inherit the information of the nesting order of the target loops as order on the rows of the transformation matrix.

5.2.1 Idea

Let us now motivate the ideas of scannable transformations informally. The central observation is that, during the iteration of one loop w inside a nest, the indices of its enclosing loops are constant, and, in general, the extent of loop w depends on all these indices.

Note that there is a potential for optimization that we do not exploit. We only exploit the information provided by the class the loop belongs to: in affine loops, we do not consider scannability, since it is a non-issue in Class 4.

Thus, let w be a non-affine loop inside a nest L and $c_1, \ldots, c_{w-1}$ the indices of its enclosing loops. Further, let $\mathcal{T}$ be a transformation matrix and w' a row with $\mathcal{T}_{w',w} \neq 0$, i.e., source dimension w is laid out in target dimension w' (at least partly, if there are multiple rows w' with $\mathcal{T}_{w',w} \neq 0$).

In order to obtain a loop nest L' which scans any possible transformed execution space of L precisely, we must require that the indices $c_1, \ldots, c_{w-1}$ of the surrounding source loops are derivable again, since these indices influence the extent of w and, thus, the extent of w'.

We name the function yielding these indices f. Note that f must express $c_1, \ldots, c_{w-1}$ in the indices $r_1, \ldots, r_{w'-1}$ of the target loops which enclose loop w'. Thus, f must not depend on indices of target loops inside loop w':

$$(\forall\ r, r' : r, r' \in \mathbb{Z}^d \wedge (\forall\ i : 1 \leq i \leq w'-1 : r_i = r'_i) : f(r) = f(r')\ )$$

Intuitively, these rules enforce that the iterations of a while loop at some level, say, w of the source loop nest are not part of some target loop (then also a while loop) at a level less than w. In other words, a while loop in the source can only be distributed across deeper levels of the target loop nest. (Compare also the theory of loop permutations [5].)

5.2.2 Formalization

The ideas of the previous section lead to the following formal definition of scannability:

Definition 10 (Scannable transformations). The transformation of a loop nest L by an invertible square matrix $\mathcal{T}$ of rank d is *scannable* iff:

$$\begin{array}{l}(\forall\ w, w' : 1 \leq w, w' \leq d \wedge \textit{naff-col}(w) \wedge \mathcal{T}_{w',w} \neq 0 : (\exists\ f : f \in \mathbb{Z}^d \rightarrow \mathbb{Z}^{w-1} : \\ \quad (\forall\ r, r', c : r, r', c \in \mathbb{Z}^d \wedge (r = \mathcal{T}\, c) \wedge (\forall\ i : 1 \leq i \leq w'-1 : r_i = r'_i) : \\ \quad\quad f(r) = (c_1, \cdots, c_{w-1})^\top = f(r')\ )))\end{array}$$

The existential quantification of f in Definition 10 makes it hard to check the scannability of a given transformation; therefore, we are interested in a more concrete condition. Not surprisingly, f is part of the inverse space-time matrix $\mathcal{T}^{-1}$. The following theorem states the precise definition of f.

Theorem 11 (Scannability test). *The transformation of a loop nest L by an invertible square matrix $\mathcal{T}$ of rank d is scannable iff:*

$$\begin{array}{l}(\forall\ w, w' : 1 \leq w, w' \leq d \wedge \textit{naff-col}(w) \wedge \mathcal{T}_{w',w} \neq 0 : \\ \quad (\forall\ r, c : 1 \leq r < w \wedge w' \leq c \leq d : \mathcal{T}^{-1}_{r,c} = 0) \wedge w \leq w')\end{array}$$

Proof. "$\Rightarrow$": We prove the two conjuncts successively.

- Left conjunct: By the definition of scannability, there is an f such that:

  $$(\forall\ r, c : r, c \in \mathbb{Z}^d \wedge (r = \mathcal{T}\, c) : f(r) = (c_1, \cdots, c_{w-1})^\top)$$

  It follows that:

  $$\begin{array}{rcl}(\forall\ r : r \in \mathbb{Z}^d : & & f(r) \\ & = & (c_1, \cdots, c_{w-1})^\top \\ & = & c|_{1,\cdots,w-1} \\ & = & (\mathcal{T}^{-1}\, r)\Big|_{1,\cdots,w-1} \\ & = & \mathcal{T}^{-1}\Big|_{1,\cdots,w-1}\ r)\end{array}$$

  f is a linear function. We name the matrix that represents it $M = \mathcal{T}^{-1}\Big|_{1,\cdots,w-1} \in \mathbb{Z}^{(w-1)\times d}$. Note that M is the upper part of $\mathcal{T}^{-1}$. By showing that the right part of M is zero, we prove that some upper right corner of $\mathcal{T}^{-1}$ is zero. The definition of scannability gives us:

  $$\begin{array}{ll} & (\forall\ r, r' : r, r' \in \mathbb{Z}^d \wedge (\forall\ i : 1 \leq i \leq w'-1 : r_i = r'_i) : f(r) = f(r')) \\ \Rightarrow & \{\ M \text{ is the matrix for } f\ \} \\ & (\forall\ r, r' : r, r' \in \mathbb{Z}^d \wedge (\forall\ i : 1 \leq i \leq w'-1 : r_i = r'_i) : M\, r = M\, r') \\ \Rightarrow & \{\ \text{definition of matrix-vector-product, ignoring equal summands}\ \} \\ & (\forall\ r, r' : r, r' \in \mathbb{Z}^d : (\forall\ i : 1 \leq i \leq w-1 : \\ & \quad (\Sigma\ j : w' \leq j \leq d : M_{i,j}\, r_j) = (\Sigma\ j : w' \leq j \leq d : M_{i,j}\, r'_j))) \\ \Rightarrow & \{\ \text{choose } r' = 0\ \} \\ & (\forall\ r : r \in \mathbb{Z}^d : (\forall\ i : 1 \leq i \leq w-1 : (\Sigma\ j : w' \leq j \leq d : M_{i,j}\, r_j) = 0)) \\ \Rightarrow & \{\ \text{arithmetic}\ \} \\ & (\forall\ i, j : 1 \leq i \leq w-1 \wedge w' \leq j \leq d : M_{i,j} = 0) \\ \Rightarrow & \{\ M = \mathcal{T}^{-1}\Big|_{1,\cdots,w-1}\ \} \\ & (\forall\ i, j : 1 \leq i \leq w-1 \wedge w' \leq j \leq d : \mathcal{T}^{-1}_{i,j} = 0)\end{array}$$

- Right conjunct: We know that $\text{rank}(\mathcal{T}^{-1}) = d$, since $\mathcal{T}$ is an invertible square matrix of rank d. Thus:

$$\begin{array}{ll} & d \\ = & \text{rank}(\mathcal{T}^{-1}) \\ \leq & \text{rank}(M) + \text{rank}(\mathcal{T}|_{w,\cdots,d}) \\ \leq & \text{rank}(M) + d - (w-1) \\ \Leftrightarrow & \{ \text{ arithmetic } \} \\ & w-1 \leq \text{rank}(M) \\ \Leftrightarrow & \{ \text{ rank}(M) \leq w-1 \text{ (since } M \text{ has } w-1 \text{ rows) } \} \\ & \text{rank}(M) = w-1 \end{array}$$

Thus, there must be some number k of non-zero columns that is at least as big as $\text{rank}(M)$. It follows that $\text{rank}(M) \leq k \leq w'-1$, since all columns from column w' to the right are zero. This yields, with the derived value for $\text{rank}(M)$, $w \leq w'$.

"$\Leftarrow$": Let the column w be a non-affine column, and let $w \leq w'$ with $\mathcal{T}_{w',w} \neq 0$. Then, let r, r', c be vectors in $\mathbb{Z}^d$ such that $r = \mathcal{T}\, c$ and $(\forall\ i : 1 \leq i \leq w'-1 : r_i = r'_i)$. Define $f(x) = \mathcal{T}^{-1}\Big|_{1,\cdots,w-1} x$. We show that this choice for f satisfies the conditions required in the definition of scannability. The right side of the if-and-only-if in Theorem 11 yields:

$$\begin{array}{ll} & (\forall\ i,j : 1 \leq i < w \wedge w' \leq j \leq d : \mathcal{T}^{-1}_{i,j} = 0) \\ \Rightarrow & \{ \ (\forall\ i : 1 \leq i \leq w'-1 : r_i = r'_i) \wedge (r = \mathcal{T}\, c) \ \} \\ & \mathcal{T}^{-1}\Big|_{1,\cdots,w-1} r = \mathcal{T}^{-1}\Big|_{1,\cdots,w-1} r' \\ \wedge & \mathcal{T}^{-1}\Big|_{1,\cdots,w-1} r = \left(\mathcal{T}^{-1} r\right)\Big|_{1,\cdots,w-1} = c|_{1,\cdots,w-1} = (c_1, \cdots, c_{w-1})^\top \\ \Leftrightarrow & \{ \text{ definition of } f \ \} \\ & f(r) = f(r') \ \wedge \ f(r) = (c_1, \cdots, c_{w-1})^\top \end{array}$$

Theorem 11 provides us with a simple way of checking whether the target space of the transformation can be scanned precisely by a target loop nest.

Let us check whether Definition 10, and thus Theorem 11, both for scannable transformations, guarantee scannable target execution spaces, i.e., whether Definition 10 is *sufficient* for creating scannable sets. In the proof of the following lemma we denote a line between two points x and y by $line(x, y)$.

Lemma 12. *The target execution space of a loop nest L obtained by a scannable and unimodular matrix $\mathcal{T}$ contains no holes.*

Proof. We prove this lemma by contradiction: assume $h = (h_1, \cdots, h_{w'}, *, \cdots, *)$ is a hole w.r.t. level w' where $*$ stands for an arbitrary value. To simplify the proof, we choose h such that the level w.r.t. its corresponding source coordinates is minimal.

$$
\begin{array}{ll}
 & (\exists\, w' \,:\, 1 \le w' \le d \,:\, h \text{ as just described}) \\
\Leftrightarrow & \{\ \text{definition of hole}\ \} \\
 & (\exists\, w' \,:\, 1 \le w' \le d \,:\, h \notin \mathcal{T}\mathcal{X} \wedge (\exists\, h^-, h^+ \,:\, h^-, h^+ \in \mathcal{T}\mathcal{X} \wedge \\
 & \;h^- = (h_1, \cdots, h_{w'-1}, h^-_{w'}, *, \cdots, *), h^+ = (h_1, \cdots, h_{w'-1}, h^+_{w'}, *, \cdots, *) \,: \\
 & \;h^+_{w'} > h_{w'} > h^-_{w'})) \\
\Rightarrow & \{\ \text{the target space is not generated by an affine loop nest}\ \} \\
 & (\exists\, w, w' \,:\, 1 \le w, w' \le d \,:\, \mathcal{T}_{w',w} \ne 0 \wedge \textit{naff-col}(w) \wedge h \notin \mathcal{T}\mathcal{X} \wedge \\
 & \;(\exists\, h^-, h^+ \,:\, h^-, h^+ \in \mathcal{T}\mathcal{X} \wedge \\
 & \;h^- = (h_1, \cdots, h_{w'-1}, h^-_{w'}, *, \cdots, *), h^+ = (h_1, \cdots, h_{w'-1}, h^+_{w'}, *, \cdots, *) \,: \\
 & \;h^+_{w'} > h_{w'} > h^-_{w'})) \\
\Rightarrow & \{\ \text{Definition 10 with } h^- \text{ as } r \text{ and } h^+ \text{ and } h \text{ as } r' \text{ and proof of Theorem 11}\ \} \\
 & (\exists\, w, w' \,:\, 1 \le w, w' \le d \,:\, \mathcal{T}_{w',w} \ne 0 \wedge \textit{naff-col}(w) \wedge h \notin \mathcal{T}\mathcal{X} \wedge \\
 & \;(\exists\, h^-, h^+ \,:\, h^-, h^+ \in \mathcal{T}\mathcal{X} \wedge \\
 & \;h^- = (h_1, \cdots, h_{w'-1}, h^-_{w'}, *, \cdots, *), h^+ = (h_1, \cdots, h_{w'-1}, h^+_{w'}, *, \cdots, *) \,: \\
 & \;h^+_{w'} > h_{w'} > h^-_{w'}) \wedge \\
 & \;\left(\mathcal{T}^{-1}\, h^+\right)\Big|_{1,\cdots,w-1} = \left(\mathcal{T}^{-1}\, h\right)\Big|_{1,\cdots,w-1} = \left(\mathcal{T}^{-1}\, h^-\right)\Big|_{1,\cdots,w-1}) \\
\Rightarrow & \{\ \mathcal{T} \text{ is injective and } h^-_{w'} \ne h_{w'} \ne h^+_{w'}\ \} \\
 & (\exists\, w, w' \,:\, 1 \le w, w' \le d \,:\, \mathcal{T}_{w',w} \ne 0 \wedge \textit{naff-col}(w) \wedge h \notin \mathcal{T}\mathcal{X} \wedge \\
 & \;(\exists\, h^-, h^+ \,:\, h^-, h^+ \in \mathcal{T}\mathcal{X} \wedge \\
 & \;h^- = (h_1, \cdots, h_{w'-1}, h^-_{w'}, *, \cdots, *), h^+ = (h_1, \cdots, h_{w'-1}, h^+_{w'}, *, \cdots, *) \,: \\
 & \;h^+_{w'} > h_{w'} > h^-_{w'}) \wedge \\
 & \;\left(\mathcal{T}^{-1}\, h^+\right)\Big|_{1,\cdots,w-1} = \left(\mathcal{T}^{-1}\, h\right)\Big|_{1,\cdots,w-1} = \left(\mathcal{T}^{-1}\, h^-\right)\Big|_{1,\cdots,w-1} \wedge \\
 & \;\left(\mathcal{T}^{-1}\, h^+\right)\Big|_{w} \ne \left(\mathcal{T}^{-1}\, h\right)\Big|_{w} \ne \left(\mathcal{T}^{-1}\, h^-\right)\Big|_{w}) \\
\Rightarrow & \{\ \text{level of } \mathcal{T}^{-1}\, h \text{ is minimal}\ \} \\
 & (\exists\, w, w' \,:\, 1 \le w, w' \le d \,:\, \mathcal{T}_{w',w} \ne 0 \wedge \textit{naff-col}(w) \wedge h \notin \mathcal{T}\mathcal{X} \wedge \\
 & \;(\exists\, h^-, h^+ \,:\, h^-, h^+ \in \mathcal{T}\mathcal{X} \wedge \\
 & \;h^- = (h_1, \cdots, h_{w'-1}, h^-_{w'}, *, \cdots, *), h^+ = (h_1, \cdots, h_{w'-1}, h^+_{w'}, *, \cdots, *) \,: \\
 & \;h^+_{w'} > h_{w'} > h^-_{w'}) \wedge \\
 & \;\left(\mathcal{T}^{-1}\, h^+\right)\Big|_{1,\cdots,w-1} = \left(\mathcal{T}^{-1}\, h\right)\Big|_{1,\cdots,w-1} = \left(\mathcal{T}^{-1}\, h^-\right)\Big|_{1,\cdots,w-1} \wedge \\
 & \;\left(\mathcal{T}^{-1}\, h^+\right)\Big|_{w} \ne \left(\mathcal{T}^{-1}\, h\right)\Big|_{w} \ne \left(\mathcal{T}^{-1}\, h^-\right)\Big|_{w} \wedge \\
 & \;(\forall\, k \,:\, w+1 \le k \le d \,:\, \left(\mathcal{T}^{-1}\, h\right)\Big|_{k} = lb_k)) \\
\Rightarrow & \{\ \text{the source loop at level } w \text{ cannot skip the index value } \left(\mathcal{T}^{-1}\, h\right)\Big|_{w}\ \} \\
 & (\exists\, w, w' \,:\, 1 \le w, w' \le d \,:\, \mathcal{T}_{w',w} \ne 0 \wedge \textit{naff-col}(w) \wedge h \notin \mathcal{T}\mathcal{X} \wedge \\
 & \;(\exists\, h^-, h^+ \,:\, h^-, h^+ \in \mathcal{T}\mathcal{X} \wedge \\
 & \;h^- = (h_1, \cdots, h_{w'-1}, h^-_{w'}, *, \cdots, *), h^+ = (h_1, \cdots, h_{w'-1}, h^+_{w'}, *, \cdots, *) \,: \\
 & \;h^+_{w'} > h_{w'} > h^-_{w'}) \wedge \\
 & \;\mathcal{T}^{-1}\, h \in \mathcal{X}) \\
\Rightarrow & \{\ \text{simplification}\ \} \\
 & h \notin \mathcal{T}\mathcal{X} \wedge \mathcal{T}^{-1}\, h \in \mathcal{X} \\
\Leftrightarrow & \{\ \text{definition of } \mathcal{X} \text{ and predicate calculus}\ \} \\
 & \textit{ff}
\end{array}
$$

Remark. Of course, scannability does not imply the validity of the space-time mapping. Take, e.g., the execution space in Figure 3.1 and the identity as the transformation. That is, leave the loops as they are, only map one of them—it does not matter which—entirely to space. This satisfies scannability, since no loops are permuted, but it violates the while dependences of that while loop mapped to space.

5.2.3 Additional Benefit of Scannable Transformations

Up to now, we have concentrated on the question of whether a set S of points is "precisely" scannable. As noted above, we intend to enumerate a superset of S and prevent the holes from execution when dealing with unscannable sets. But we must still find a loop nest, i.e., loop bounds—in this case, to enumerate the superset. The following example shows that this is, in general, a non-trivial task.

Example 8. Take the loop nest

for $i := 0$ to n do
  for $j := 0$ while $condition(i,j)$ do
    body
  enddo
enddo

and try to interchange the loops, i.e.,

$$\mathcal{T} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

The bound for the outer target loop will always have to compute the maximal extent of all $n+1$ while loop instances; but this requires both indices i and j, since we have to evaluate the conditions $condition(i,j)$ for all indices i and j. So, there is no precise outer loop bound that does not depend on the inner indices.

Thus, there cannot exist a generalization of the Fourier-Motzkin elimination method for arbitrary loop nests with arbitrary transformations, which yields target loop bounds enumerating (even some superset of) the target execution space and only depending on outer loop indices and parameters.

This raises the question: is it possible to find (precise) loop bounds for the target execution space generated by a scannable transformation which do not depend on inner indices? The answer is given by the following immediate consequence of Definition 10.

Lemma 13. *The bounds of the target loops which enumerate the target execution space generated by a scannable transformation do not depend on loop indices of inner target loops.*

Proof. In the source program there exists some (not explicitly given) function $g_w(c_1, \cdots, c_{w-1})$, which yields the lower (upper) bound b_w of a source loop w for fixed source indices $(c_1, \cdots, c_{w-1})$. We show that the lemma is true for the target loop bound at any level w'. Therefore, let D be the set of all source dimensions w which are (partly) laid out in target dimension w'. Thus, for any w':

$$
\begin{array}{ll}
& (\forall\, w \,:\, w \in D \,:\, \mathcal{T}_{w',w} \neq 0) \wedge \mathcal{T} \text{ scannable} \\
\Rightarrow & \{ \text{ Definition 10 } \} \\
& (\forall\, w : w \in D : (\exists\, f : f \in \mathbb{Z}^d \rightarrow \mathbb{Z}^{w-1} : \\
& \quad (\forall\, r, r', c : r, r', c \in \mathbb{Z}^d \wedge (r = \mathcal{T}\, c) \wedge (\forall\, i : 1 \leq i \leq w'-1 : r_i = r'_i) : \\
& \qquad\qquad f(r) = (c_1, \cdots, c_{w-1}) = f(r') \;)) \vee \neg \textit{naff-col}(w)) \\
\Rightarrow & \{ \text{ define } b_w := g_w(c_1, \cdots, c_{w-1}); \text{ insert it as condition and as consequence } \} \\
& (\forall\, w : w \in D : (\exists\, f : f \in \mathbb{Z}^d \rightarrow \mathbb{Z}^{w-1} : (\forall\, r, r', c : r, r', c \in \mathbb{Z}^d \wedge (r = \mathcal{T}\, c) \wedge \\
& \quad g_w(c_1, \cdots, c_{w-1}) = b_w \wedge (\forall\, i : 1 \leq i \leq w'-1 : r_i = r'_i) : \\
& \qquad\qquad f(r) = (c_1, \cdots, c_{w-1}) = f(r') \wedge g_w(c_1, \cdots, c_{w-1}) = b_w \;)) \vee \neg \textit{naff-col}(w)) \\
\Rightarrow & \{ \text{ substitute } (c_1, \cdots, c_{w-1}) \text{ by } f(r) \text{ and } f(r') \;\} \\
& (\forall\, w : w \in D : (\exists\, f : f \in \mathbb{Z}^d \rightarrow \mathbb{Z}^{w-1} : (\forall\, r, r', c : r, r', c \in \mathbb{Z}^d \wedge (r = \mathcal{T}\, c) \wedge \\
& \quad g_w(c_1, \cdots, c_{w-1}) = b_w \wedge (\forall\, i : 1 \leq i \leq w'-1 : r_i = r'_i) : \\
& \qquad\qquad f(r) = (c_1, \cdots, c_{w-1}) \wedge g_w(f(r)) = g_w(f(r')) = b_w \;)) \vee \neg \textit{naff-col}(w)) \\
\Rightarrow & \{ \text{ omit } f(r) = (c_1, \cdots, c_{w-1}); \text{ substitute } f \text{ by } \mathcal{T}^{-1}\Big|_{1,\cdots,w-1} \text{ (cf. proof of Theorem 11) } \} \\
& (\forall\, w : w \in D : (\forall\, r, r', c : r, r', c \in \mathbb{Z}^d \wedge (r = \mathcal{T}\, c) \wedge \\
& \quad g_w(c_1, \cdots, c_{w-1}) = b_w \wedge (\forall\, i : 1 \leq i \leq w'-1 : r_i = r'_i) : \\
& \qquad\qquad g_w(\mathcal{T}^{-1}\Big|_{1,\cdots,w-1} r) = g_w(\mathcal{T}^{-1}\Big|_{1,\cdots,w-1} r') = b_w \;) \vee \neg \textit{naff-col}(w))
\end{array}
$$

Thus, any two points r, r' which do not differ in outer target loop indices compute the same border coordinate for the target loop w' with fixed outer indices $(r_1, \cdots, r_{w'-1})$.

Remark. Note that Lemmas 12 and 13 are implications only. In both cases, the reverse implication is not true since, e.g., for convex loop nests the target space is always scannable, regardless of the transformation.

5.2.4 Applicability

5.2.5 Choices of Space-Time Mappings

Our requirements for a precise scan limit the choice of space-time mapping significantly. Let us discuss what freedom of choice is left:

- If only the outermost loop of the nest is a non-affine loop, then every space-time mapping produces scannable execution spaces, since the scannability condition is trivially satisfied ($1 \leq r < w$ is impossible for $w = 1$).
- In a two-dimensional nest with an inner non-affine loop, the invertible space-time matrix, and, equivalently, its inverse, must have the form $\begin{pmatrix} x & 0 \\ y & z \end{pmatrix}$ with $y \in \mathbb{Z}$ and $x, z \in \mathbb{Z} \backslash \{0\}$.
- For deeper loop nests, there is a wider choice of space-time mappings. It is easy to show that all lower triangular matrices are scannable; however, this is not a necessary condition. Assume a nest of three loops of which only the second is a `while` loop. Then, the following space-time matrix is scannable:

$$
\mathcal{T} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 1 & 2 \end{pmatrix} \qquad \mathcal{T}^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{pmatrix}
$$

5.2.6 Asynchronous Target Loop Nests and Scannability

Now that we have demonstrated the benefits of scannable transformations, we want to know whether there always exists a scannable transformation. Since it depends on the position of the schedule in the space-time matrix, the answer is different for synchronous and asynchronous programs.

Lemma 14. *For asynchronous target loop nests, a scannable space-time mapping can always be found.*

Proof. Let $\mathcal{T}$ be the identity matrix of rank d, where d is the depth of the loop nest. $\mathcal{T}$ is both scannable (Definition 10) and a valid allocation since we imposed no requirements on allocations (Definition 7), and thus, by appending rows for the schedule dimensions, we obtain a valid space-time matrix.

Note that the identity is not the only allocation which leads to a scannable space-time mapping—it is just the simplest and most general one for the proof. Another very similar, scannable and always valid asynchronous space-time matrix can be composed as follows: the first rows, representing the allocation, are the unit vectors of length d for dimensions $1, \cdots, d-1$, and the row(s) for the schedule is/are appended below. If the schedule is one-dimensional, the resulting square matrix represents a skewing of all loops into the innermost dimension, which represents time. Of course, one may choose different allocations in practice.

5.3 Unscannable Execution Spaces

5.3.1 Motivation: Why Unscannable Transformations?

One might wonder whether it is necessary to consider unscannable transformations at all. Unfortunately, the answer is yes—if one is interested in synchronous target loop nests, i.e., nests whose outer loop is sequential.

Consider some while loop in the source loop nest but not at the outer level. Because of the while dependences, every while loop must be partially laid out in time. But time is the outer target loop, so portions of the while loop must move to an outer level—a violation of the scannability condition! Thus, only the trivial case of a for loop nest with an enclosing while loop can have a synchronous target loop nest that satisfies scannability.

5.3.2 Controlling the Scan of an Unscannable Execution Space

To generate target code for an unscannable target execution space we must enumerate a superset of it. We name this superset $\mathcal{TS}$ and its inverse image under the space-time mapping $\mathcal{S}$.

For loops of Class 2 the source loop bounds are given as arithmetic expressions which can be evaluated at any point. Therefore, one can, separately for every point, determine whether the point belongs to $\mathcal{TX}$ or to $\mathcal{TS} \backslash \mathcal{TX}$.

In while loops the upper bound is not given explicitly but calculated iteratively instead. Thus, the information about the termination of a while loop can only be propagated along the tooth of the while loop. Consequently, at a point in $\mathcal{TS}$, one cannot decide by local information only whether the point belongs to $\mathcal{TX}$ or not, but one needs the information about the termination of its enclosing while loops.

For this purpose we define a predicate for a nest of while loops which is an accurate recognizer of the points in $\mathcal{TX}$, i.e., which distinguishes the points in $\mathcal{TX}$ from those outside. In the following chapters, we use this predicate to prevent the execution of holes in the target polyhedron at run time.

Note that for simplicity we only consider while loops in the loop nest; we do not consider possible additional for loops in the following discussions since they only introduce additional dimensions but do not raise any problems.

Definition 15 (Activity recognizer $active_r$ **and** $active$**).** Let r be some level of the source loop nest and w the while loop at that level. $active_r$ holds for any point x in $\mathcal{I}$ iff the source program enumerates x, that is, iff at least the while condition $condition_r$ of loop w is evaluated at point x. Formally:

$$
\begin{array}{llll}
(\forall \ (x_1,\cdots,x_d) : (x_1,\cdots,x_d)\in\mathcal{I} : (\forall\, r : 1\le r\le d : active_r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d) = \\
\quad \mathsf{if} \ \ x_r > lb_r & \rightarrow & active_r(x_1,\cdots,x_r-1,lb_{r+1},\cdots,lb_d)\wedge & (1)\\
 & & \qquad condition_r(x_1,\cdots,x_r-1) & \\
\quad [\,] \ \ x_r = lb_r \wedge r>1 & \rightarrow & active_{r-1}(x_1,\cdots,x_{r-1},lb_r,\cdots,lb_d)\wedge & (2)\\
 & & \qquad condition_{r-1}(x_1,\cdots,x_{r-1}) & \\
\quad [\,] \ \ x_r = lb_r \wedge r=1 & \rightarrow & tt & (3)\\
\quad [\,] \ \ x_r < lb_r & \rightarrow & f\!f & (4)\\
\quad \mathsf{fi} \ \ )) & & &
\end{array}
$$

$$active(x_1,\cdots,x_d) = (\exists\, r : 1\le r\le d : active_r(x_1,\cdots,x_d))$$

The cases of the defining equation can be explained as follows. Point $(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)$ is active with respect to level r iff

(1) the point represents some non-leading step of a loop, the while condition holds and the previous step is active with respect to level r (hence the $x_r - 1$), or

(2) the point represents the first step of an inner loop, the while condition holds for the immediately enclosing loop and the point is active with respect to the level of the immediately enclosing loop (hence the x_{r-1}), or

(3) the point represents the first step of the entire loop nest.

In all other cases, $(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)$ is inactive with respect to level r. These include the case where the while condition of w is violated (covered by alternatives (1) and (2)), and the case that the point is not even in the index space (alternative (4)). Note, that points for which the while condition holds at level r but not at level $r+1$ are active with respect to level r but not with respect to levels $r+1$ and deeper.

The recursive definition of predicate $active_r$ follows the dependences which are introduced by the while indices. Since our space-time mapping must respect these dependences, we can be sure that, during scanning, the activity of any point x in $\mathcal{TI}$ need not be checked before the activity of its predecessor has been checked. Therefore, we can compute predicate $active_r$, for every point on every tooth of the execution comb, in sequence from the root to the tip and store the result until it is needed. Note that $active_r$ at point $(x_1, \cdots, x_r, lb_{r+1}, \cdots, lb_d)$ is calculated from $condition_r(x_1, \cdots, x_r, lb_{r+1}, \cdots, lb_d)$ or $condition_{r-1}(x_1, \cdots, x_{r-1}, lb_r, \cdots, lb_d)$, which is usually data dependent on the loop body at $(x_1, \cdots, x_r, lb_{r+1}, \cdots, lb_d)$ or $(x_1, \cdots, x_{r-1}, lb_r, \cdots, lb_d)$, respectively. In this case, the computation of the values of $active_r$ is executed alternately with the computations of the loop body.

Note that $active_r$ at point $(x_1, \cdots, x_r, lb_{r+1}, \cdots, lb_d)$ depends on $active_r$ at exactly one other point (see the definition); $active_r$ at point $(x_1, \cdots, x_r, lb_{r+1}, \cdots, lb_d)$ is used for the computation of $active_r$ at point $(x_1, \cdots, x_r{+}1, lb_{r+1}, \cdots, lb_d)$ and, if r is not the innermost loop level, also for the computation of $active_{r+1}$ at itself.

Since the index space of a while loop nest contains points that do not model a loop step but only a terminating test, we also require a recognizer, *executed*, for points of $\mathcal{I}$, that do represent the execution of the loop body.

Definition 16 (Recognizer $executed_r$ **and** *executed***).**

$$(\forall\, r : 1 \leq r \leq d : (\forall\, x : x \in \mathcal{I} : executed_r(x_1, \cdots, x_r, lb_{r+1}, \cdots, lb_d) \Leftrightarrow \\ (active_r(x_1, \cdots, x_r, lb_{r+1}, \cdots, lb_d) \wedge condition_r(x_1, \cdots, x_r))))$$

$$(\forall\, x : x \in \mathcal{I} : executed(x) \Leftrightarrow executed_d(x_1, \cdots, x_d))$$

At this point, we have the machinery for a formal definition of the execution space:

Definition 17 (Execution space). $\mathcal{X} = \{x \in \mathcal{I} : executed(x)\}$

Later on, we shall need its extension to all points that are active in some dimension:

Definition 18 (Activity space). $\widehat{\mathcal{X}} = \{x \in \mathcal{I} : active(x)\}$

Some hints on the implementation of the introduced predicates *executed* and *active* are given in Chapter 7 which treats the problem of termination detection, since we want to integrate the solutions for the termination and the scanning problem in one common scheme.

5.4 The Example

Let us consider the space-time mappings of Table 3.3 on page 25.

The essential transformation for statement S_1 is the identity matrix of dimensionality 1. It is trivially scannable. The same is true for the other one-dimensional statements S_2 and S_3, since constant offsets do not become part of the essential transformation matrix.

Analogously, the two-dimensional statements S_5 to S_7 and S_9 have identical essential transformations. Let us first check the scannability of the synchronous transformation matrix:

$$\mathcal{T} = \begin{pmatrix} 1 & 4 \\ 1 & 0 \end{pmatrix} \qquad \mathcal{T}^{-1} = \frac{1}{4}\begin{pmatrix} 0 & 4 \\ 1 & -1 \end{pmatrix}$$

Applying Theorem 11 for $w = 2$ and $w' = 1$ shows that this transformation is not scannable, as is to be expected following the explanations in Section 5.3.1. We postpone the presentation of the target code for this case to Chapter 7, where the rest of the necessary theory will be presented.

Let us now check the scannability of the asynchronous transformation matrix for the one-dimensional allocation:

$$\mathcal{T}' = \begin{pmatrix} 1 & 0 \\ 1 & 4 \end{pmatrix} \qquad \mathcal{T}'^{-1} = \frac{1}{4}\begin{pmatrix} 4 & 0 \\ -1 & 1 \end{pmatrix}$$

Theorem 11 is trivially satisfied for $w = 1$. For $w = 2$, the only non-zero entry in $\mathcal{T}'$ is in row $w' = 2$; since $\mathcal{T}'^{-1}_{1,2} = 0$ the condition is satisfied, too. Thus: $\mathcal{T}'$ is scannable.

For the three-dimensional statement S_8 we get the same result.

Consequently, for every statement, there is an asynchronous loop nest which scans precisely the target execution space of this statement.

Since we are now sure of its existence, let us try to find an asynchronous loop "nest" for some one-dimensional statement, say, S_2. In the asynchronous case the outermost loop (in this one-dimensional case the only target loop) is a loop in space; we name its index p. The allocation of S_2 yields $p = n$. Thus, we enumerate the target execution space of S_2—analogously to the source execution space—with for $p := 0$ while $node[p] \neq \perp$ do S_2.

On the other hand, this raises a big problem: since we do not know at compile time the extent of the while loop, we must allocate infinitely many processors initially. This problem of while loops in space is tackled in Chapter 6.

Therefore, we also postpone the presentation of the target code of the example program under the scannable transformation with the one-dimensional allocation until the end of Chapter 6.

Chapter 6

Processor Allocation

An important problem of parallelizing general loop nests is the determination of upper bounds for all target loops. In this chapter we address the problem of bounding the space dimensions whereas the next chapter deals with the bounds on the time dimensions.

The problem of processor allocation is treated in two phases: first, we establish whether the processor space can be limited at compile time at all, and second, we make some remarks on partitioning/tiling techniques.

6.1 Limitation of the Processor Dimensions

Since we allow the upper loop bound to be unknown, the space-time mapping may be defined on an infinite domain (index space) and, thus, may define an infinite range (target space). It is easy to ascertain that only a finite number of processors will be required at any point in time. We can state this fact as a theorem. Since only the while loops contribute to the infinity of the index space, we do not consider for loops but show only that any nest of while loops defines, at any time step, a finite set of processors in the target space. Then, we conclude without further proof that mixed loops also do so.

Theorem 19. *Let $v_1, \ldots, v_r$ be linearly independent vectors of $\mathbb{Z}^r$ and $\alpha_1, \ldots, \alpha_r \in \mathbb{N} \backslash \{0\}$. Then the intersection of any hyperplane H through the set of points $\{(\alpha_1\, v_1, 0, \ldots, 0), \ldots, (0, \ldots, 0, \alpha_r\, v_r)\}$ and the polyhedral cone K spanned by the vectors $v_1, \ldots, v_r$ is finite.*

Proof. Our basis of $\mathbb{Z}^r$ is $\{v_1, \ldots, v_r\}$. Then $K = \{x \mid x \in \mathbb{N}^r \wedge -I\, x \leq 0\} = \mathbb{N}^r$ is the polyhedral cone spanned by $v_1, \ldots, v_r$ [44]. (I is the identity matrix.) Furthermore, $H = \{x \mid x \in \mathbb{Z}^n \wedge (\frac{1}{\alpha_1}, \ldots, \frac{1}{\alpha_r})\, x = 1\}$. Then:

$$\begin{aligned} H \cap K &= \{x \mid x \in \mathbb{N}^r \wedge (\Sigma\, i : 0 < i \leq r : \frac{x_i}{\alpha_i}) = 1\} \\ &\subseteq \{x \mid x \in \mathbb{N}^r \wedge (\forall\, i : 0 < i \leq r : 0 \leq x_i \leq \alpha_i)\} \end{aligned}$$

Since the superset on the right is finite, so is $H \cap K$.

Corollary 20 (Finiteness of time slices). *In the polyhedron model, the iteration space $\mathcal{I}$ representing a nest of loops is the cone K, and $H \cap K$ corresponds to some time slice $t^{-1}(x) \subset \mathcal{I}$ for a fixed $x \in t(\mathcal{I})$. Thus, each time slice is finite.*

However, this corollary does not specify an upper bound on the number of processors. We know that the number of processes is given by an affine function of time, i.e., the number of used processors grows affinely with time. But, for asynchronous loop nests, the time coordinate is enumerated by the inner loops and, thus, cannot be used in the bounds of the outer spatial loops. As we have seen in Section 5.4, we would have to allocate infinitely many processors initially. Collard [14] solves this problem for the case that there is one while loop at the outermost level.

In a real implementation the unboundedness must be solved at compile time since, in general, all processors must be allocated before the parallel program starts its execution. This can be achieved by standard partitioning or folding techniques (cf. Section 6.2).

Remark. The usual practice of allocating processors at the start of a program's execution might be taken as an explanation for the absence of a parwhile construct (a parallel while loop with an upper bound given by an arbitrary boolean expression). But there is also a theoretical reason: the construct parwhile would have to activate a set of processors in one time step (like parfor) and would, therefore, have to test all its conditions successively until the first termination condition evaluates to *tt*; this cannot be done in constant time.

6.2 Partitioning Techniques

Laying out a while loop partly in space only makes sense if we bound the number of processors required by partitioning the processor space in some way. This has become an active area of research recently [18, 57, 58].

The idea of *partitioning* is that a single dimension can also be enumerated by a nest of loops, not only by a single loop. To apply this idea to a polyhedron $\mathcal{P}$ we proceed in several steps: first, we select the dimensions which shall be partitioned (let us denote the polyhedron projected on these dimensions by $\overline{\mathcal{P}}$); second, we define a tile, i.e., a polytope with fixed shape and size in $\overline{\mathcal{P}}$; third, we generate nested loops enumerating all points of the tile and all tiles necessary to cover $\overline{\mathcal{P}}$; fourth, we replace the original loops enumerating the selected dimensions of $\mathcal{P}$ by the new loop nest.

In our framework we want to partition the dimensions in (virtual) space, computed by the allocator, and replace them by dimensions in real space, i.e., dimensions enumerating real processors, and dimensions in time. These dimensions in time are in addition to the time dimensions enumerating the schedule. In other words, partitioning offers us a trade-off between space and time. Note that we partition the target space, not the source space as is typical in literature.

Due to the degrees of freedom left, there are two contrary ways of partitioning which are known as *LSGP* (locally sequential, globally parallel) and *LPGS* (locally parallel, globally sequential) [38]. In the LSGP method, the points inside a tile are enumerated sequentially by one processor (locally sequential) and the tiles are distributed among the processors (globally parallel), i.e., one uses one processor per tile. In the LPGS method, the tile corresponds to the real set of processors, i.e., every processor is responsible for one point of the tile (locally parallel), and the tiles are enumerated successively (globally sequential).

Recent literature prefers the LSGP method [18] since, in general, there are many local communications which become obsolete if neighboring operations are executed on one processor. Additionally, there are effective methods for choosing the shape of the tile according to the dependences of the polyhedron to be scanned, which results in a further reduction of communication.

However, in the presence of while loops we cannot choose the LSGP method since we cannot predict the "global" size and, therefore, the extent of the "globally parallel" dimensions. We must use LPGS partitioning methods; they yield constant bounds for the processor dimensions and map all unbounded dimensions to time. The only remaining problem for parallelizing loop nests containing while loops is how to handle the termination of the target loops in time. This will be discussed in the next chapter.

Remark 21 (Parallel loops). We have just seen that, due to the application of LPGS partitioning for asynchronous loop nests, the loops in space are for loops. In Section 6.1 we have learned that in synchronous loop nests we can bound the space dimensions by expressions in the indices of surrounding loops in time (Corollary 20). Thus, we can make the following observation: in the target loop nest, every loop in space is a for loop (thus a parfor), even if there are while dimensions mapped (partly) to this space dimension.

Remark 22 (Code generation). Note that the partitioning techniques introduce additional loops in time. Therefore, we must take care that these additional loops respect the schedule (remember that the execution order of sequential loops is determined by the lexicographic order of the index vectors): if the additional loops in time are nested inside the loops enumerating the schedule then the schedule's index determines the execution order—the additional time dimensions are only a refinement of the schedule. However, if the additional loops in time are nested outside of the loops enumerating the schedule then the additional loops determine the execution order, i.e., the schedule is not respected any more, which leads to an incorrect target loop nest!

Since, first, the partitioning method replaces the original spatial loops by the nest of new loops in space and time, and, second, the new loops in time must be inner loops w.r.t. the dimensions of the schedule, the original spatial loops must be inner loops w.r.t. the schedule. In other words, loop nests which are subject to a partitioning must specify synchronous parallelism.

Note that taking the synchronous program as input for partitioning is a sufficient but not a necessary condition for respecting the schedule; the application to the example program in the next section starts with the asynchronous program and yields a correct target program.

Note, in addition, that the code after partitioning as just described is synchronous. However, the dimensions of the real processors are bounded by expressions describing the real parallel machine, i.e., these dimensions are bounded by parameters known at compile time. If these are the only expressions in the bounds of the parallel loops, e.g., if there are no expressions depending on outer loop indices, then we can easily shift these parallel loops to the outermost levels (even without Fourier-Motzkin elimination). This shift results in an asynchronous target program.

Otherwise, it is also possible to obtain an asynchronous target program, by first ignoring the additional bounds, subsequently performing the shift and finally introducing guards which prevent those points from execution which are additionally enumerated because of ignoring the additional bounds. We do not go into more detail here since this is independent of whether the loops being while loops or for loops; details can be found in [56].

```
terminated := ff;
parfor pp := 0 to NrProc−1 do
   for tp := 0 while ¬terminated step NrProc do
      p := tp+pp;
      if ¬terminated then
         if node[p] = ⊥ then terminated := tt endif
      endif
      if ¬terminated then
         body(p)
      endif
   enddo
enddo
```

Figure 6.1: A single while loop (partly) in space after partitioning

6.3 The Example

Let us first partition the one-dimensional loop (nest) for $p := 0$ while $node[p] \neq \perp$ do $body(p)$ of Section 5.4.

We use as processor layout a one-dimensional array of $NrProc$ processors. In the partitioned program (Figure 6.1) the for loop with index pp enumerates the $NrProc$ (i.e., a constant number of) "locally parallel" processors, whereas the while loop with index tp, laid out in time, enumerates the tiles "globally sequentially". For simplicity we keep the original index p throughout the body; its value is computed by the first statement of the new loop body.

Note that the original termination condition is treated as a regular statement and is therefore located in the body of the loop.

Both, the necessity of partitioning and the fact that termination conditions become regular statements in the loop body, have an unavoidable consequence: the original loop body must be guarded. On the other hand, guards in the body of the target loop nest occur anyway if one deals with by-statement transformations or piecewise affine functions, as we have seen in Section 3.3.2. For simplicity we decided to guard every separate statement in the loop body individually with all necessary conditions instead of using a nest of guards—even if some parts of the guards apply to several statements.

Note that the loop nest in Figure 6.1 is not complete: there is no dimension enumerating the time t_1 computed by the scheduler. As announced in the previous section, we want to generate a partitioned version of the asynchronous program. However, if we nest the dimension of the schedule inside the additional time dimension tp which is caused by partitioning, then we are modifying the schedule. Therefore, we must convince ourselves that this modification preserves validity: intuitively, the new schedule (tp, t_1) enforces that every processor first terminates the tooth which it is currently working on, before starting a new tooth with a larger value for tp. Since there are no dependences from any tooth to one of its predecessor teeth, this new schedule is also valid. The formal proof has to establish that the new schedule (tp, t_1) respects every dependence; we omit it here.

Now we are able to present the target code, which is given in Figure 6.2. The basic structure is equivalent to the one of Example 5 on page 26. The main difference is that the guards have an additional conjunct, due to partitioning, and the fact that the termination condition is evaluated inside the loop. The initializations and the loop header are taken from Figure 6.1. The only modification is in the computation of predicate *terminated*, which results from the fact that the code in Figure 6.2 is an executable function for distributed-memory machines, on which the old value of *terminated* must be received and its new value must be sent explicitly. This is done by the blocking communication primitives SendNode and ReceiveNode which, similarly to the corresponding PARIX command: take as first argument the number of a real processor and as second argument the value to be sent or received. (The flag *detector* and the conditional ReceiveNode statement at the bottom of the outermost loop are only necessary due to the blocking communications.)

Of course, the target program of Figure 6.2 can be optimized a lot. For example, we need not store the value of *tag* at every point in a separate variable but we could use array *tag* itself. However, the goal of this example is to show how the methods described so far can derive a parallel loop nest from a sequential loop nest containing while loops.

```
terminated := ff
parfor pp := 0 to NrProc−1 do
  for tp := 0 while not terminated step NrProc do
    p := tp+pp
    ub_d[p] := ∞
    for t_1 := p while t_1 ≤ max(p+1, p+4ub_d[p]+8) do
      if not terminated and p = t_1 then
        if p>0 then ReceiveNode((p−1)%NrProc, terminated) endif
        if not terminated and node[p] = ⊥ then
          terminated := tt
          detector := tt
        endif
        SendNode((p+1)%NrProc, terminated)
      endif
      if not terminated and p+1 = t_1 then
        rt[p, 0] := p
        nxt[p] := 1
      endif
      if not terminated and (p+4) ≤ t_1 < (p+4ub_d[p]+4) and
                    (t_1−p−4)%4 = 0 then
        if rt[p, (t_1−p−4)/4] = ⊥ then ub_d[p] := (t_1−p−4)/4 endif
      endif
      if not terminated and (p+5) ≤ t_1 < (p+4ub_d[p]+5) and
                    (t_1−p−5)%4 = 0 then
        if_cond[p, (t_1−p−5)/4] := not tag[p, rt[p, (t_1−p−5)/4]]
      endif
      if not terminated and (p+6) ≤ t_1 < (p+4ub_d[p]+6) and
                    (t_1−p−6)%4 = 0 and if_cond[p, (t_1−p−6)/4] then
        tag[p, rt[p, (t_1−p−6)/4]] := tt
        ub_c[p, (t_1−p−6)/4] := nrsuc[rt[p, (t_1−p−6)/4]]
      endif
      if not terminated and (p+8) ≤ t_1 < (p+4ub_d[p]+8) and
                    (t_1−p−8)%4 = 0 and if_cond[p, (t_1−p−8)/4] then
        nxt[p] := nxt[p] + nrsuc[rt[p, (t_1−p−8)/4]]
      endif
      if not terminated and (p+7) ≤ t_1 < (p+4ub_d[p]+7) and
                  (t_1−p−7)%4 = 0 and if_cond[p, (t_1−p−7)/4] then
        for t_2 := 0 to ub_c[p, ⌊(t_1−p−7)/4⌋] − 1 do
          rt[p, t_2+nxt[p]] := suc[rt[p, (t_1−p−7)/4], t_2]
        enddo
      endif
    enddo
  enddo
  if detector then ReceiveNode((p−1)%NrProc, terminated) endif
enddo
```

Figure 6.2: Target program for the scannable transformation with one-dimensional allocation

Chapter 7

Termination Detection

So far, we have described methods for preventing holes inside a scanned target space from execution and we bound loops in space by partitioning. The remaining open question is: how do we bound the loops in time? As in the previous chapters we assume that the source program terminates; still, esp. for unscannable target execution spaces, it is a difficult problem to find bounds for the loops in time.

Example 9. Take again the loop nest

for $i := 0$ to n do
  for $j := 0$ while $condition(i, j)$ do
    body
  enddo
enddo

and as space-time mapping

$$\begin{pmatrix} t \\ p \end{pmatrix} = \begin{pmatrix} j \\ i \end{pmatrix},$$

where p is space and t is time. (We assume that this transformation respects the dependences of the body; the while dependence is respected.) With this mapping there is no elegant way of expressing the termination condition of the outermost loop. As stated in Example 8 on page 41, we have to evaluate the conditions $condition(i, j)$ for all i and j, i.e., we need both indices. A possible termination condition would be

$$(\forall\, p \,:\, 0 \leq p \leq n \,:\, (\exists\, t' \,:\, 0 \leq t' \leq t \,:\, \neg condition(p, t'))).$$

These quantifications are potentially costly because, in general, their ranges grow with time.

The common idea behind all options discussed in the succeeding sections is: we terminate the execution as soon as we recognize that there is no more activity in the scanned space. Each of the following sections proposes a different way for determining this fact by interpreting and detecting "no activity", depending on the target language and the target architecture.

7.1 Termination Detection for Special Languages

Some data-parallel languages provide support for detecting distributed termination. A good example is the construct whilesomewhere in Hyper-C [35]. This parallel loop construct takes

$executed(x_1, \cdots, x_d) \equiv$
  $r := level(x_1, \cdots, x_d)$;
  if $exec_r[x_1, \cdots, x_{r-1}, x_r - 1] \wedge \neg condition_r(x_1, \cdots, x_r)$ then
    $decr(counter)$
  endif ;
  $exec_r[x_1, \cdots, x_r] := exec_r[x_1, \cdots, x_{r-1}, x_r - 1] \wedge condition_r(x_1, \cdots, x_r)$;
  for $k := 1 + r$ to d do
    $exec_k[x_1, \cdots, x_k] := exec_{k-1}[x_1, \cdots, x_{k-1}] \wedge condition_k(x_1, \cdots, x_k)$;
    if $exec_k[x_1, \cdots, x_k]$ then $incr(counter)$ endif
  enddo ;
  barrier ;
  $terminated := (counter = 0)$;
  barrier ;
  return $(exec_d[x_1, \cdots, x_d])$

Figure 7.1: Formalization of the counter scheme

as parameter a boolean function b which is evaluated at every processor; the loop bounded by whilesomewhere terminates iff all processors evaluate function b to ff. For synchronous target loop nests with only one dimension in time, this construct can be used directly to bound the loop in time.

The idea is as follows. The execution of a loop nest containing while loops terminates when all processors are inactive according to Definition 15. So, the loop in time can be bounded by "whilesomewhere *active*". This solves the termination detection problem.

In the following sections we present two termination detection algorithms, both for shared and one of them for distributed memory systems, in the case that the target language used does not support termination detection directly. Note that there are a lot of general termination detection algorithms, but these are not of interest to us since we are in the fortunate position that we know a lot about the structure of the program parts for which we want to detect termination.

We want to find a predicate *terminated* which can be used as a termination condition of the while loops in time. Thus, the goal of the next sections is to find (implementable) definitions for this predicate.

7.2 Termination Detection in Shared Memory

7.2.1 Idea

The execution of a while loop nest terminates when the outermost while loop has terminated and all instances of inner while loops have terminated, too—in other words, when all teeth have terminated. To implement this, we use a shared global *counter* that is incremented at the root and decremented at the tip of every tooth in any dimension. Thus, the whole program terminates if and only if there are no active teeth left, i.e., the counter has been reset to 0.

```
Algorithm executed_generator
Input:
    • The d while loop conditions.
    • The d loop counters (x_1, ···, x_d) (become the arguments to executed).
Output: Code implementing function executed.

generate( function executed(x_1, ···, x_d) : boolean )
for r:=d downto 0
    if r≥1 then
        generate( if x_r > lb_r then  )
        generate( if exec_r[x_1, ···, x_{r-1}, x_r -1] and not condition_r(x_1, ···, x_r)
                  then decr(count) endif )
        generate( exec_r[x_1, ···, x_r] := exec_r[x_1, ···, x_{r-1}, x_r -1] and
                  condition_r(x_1, ···, x_r) )
    end if
    for k := r+1 to d
        generate( exec_k[x_1, ···, x_k] := exec_{k-1}[x_1, ···, x_{k-1}] and
                  condition_k(x_1, ···, x_k) )
        generate(if exec_k[x_1, ···, x_k] then incr(count) endif )
    end for
    if r≥1 then generate ( else ) else generate ( endif )
end for
generate( barrier )
generate( terminated := (count = 0) )
generate( barrier )
generate( return (exec_d[x_1, ···, x_d]) )
```

Figure 7.2: Algorithm *executed_generator* for automatic generation of the code for *executed*

7.2.2 Formalization

A formalization of this idea can be added to an imperative specification of *executed* such that the calculation of *terminated* is hidden as a side effect of the masking function *executed* in the target program ($exec_r$ is an r-dimensional persistent array that stores the value of $executed_r(x_1, \cdots, x_r, lb_{r+1}, \cdots, lb_d)$). Function *executed* is called with the source coordinates of each scanned point in the target index space.

The specification of function *executed* is presented in Figure 7.1, where functions $incr(counter)$ and $decr(counter)$ atomically increment and decrement *counter*, respectively. $condition_0()$ and $executed_0()$ must be initialized to tt. The *level* of a point is defined as d minus the number of trailing lb coordinates.

If we expand the definition of *level* and unroll the loop on k at compile time, we obtain the code generation scheme for *executed* in Figure 7.2. The code generated for *executed* in the case of two nested while loops is given in Figure 7.3.

Various instances of *executed* interact as follows. At every time step t, function *executed* is called on every processor p of $\mathcal{TS}$, i.e., on the entire hyperplane t, intersected with $\mathcal{TS}$, to check whether the transformed body at the coordinates (t, p) must be executed or not. Essentially, this check boils down to the evaluation of the while conditions. The combination of all these evaluations determines whether, at time t, the program terminates or not, i.e., whether the value of *count* is zero. Of course, it is mandatory that every processor has the same view of the state of global termination at every logical time t (otherwise, it could perhaps stop too early and block the entire computation). For this reason, we must ensure that all updates of the counters (esp. all increments) in the various instances of *executed* have completed before any processor reads the value of *count*. In addition, we must ensure that no processor can start its next iteration, and possibly modify the counter, before all other processors have read *count*. Both cases can only be guaranteed by barrier synchronization.

```
function executed(w1, w2) : boolean
if w2 > lb2 then
   if exec2[w1, w2−1] and not P2(w1, w2) then decr(count) endif ;
   exec2[w1, w2] := exec2[w1, w2−1] and P2(w1, w2) ;
else if w1 > lb1 then
   if exec1[w1−1] and not P1(w1) then decr(count) endif ;
   exec1[w1] := exec1[w1−1] and P1(w1) ;
   exec2[w1, w2] := exec1[w1] and P2(w1, w2) ;
   if exec2[w1, w2] then incr(count) endif
else /* w1 = lb1, w2 = lb2 */
   exec1[w1] := P1(w1) ;
   if exec1[w1] then incr(count) endif ;
   exec2[w1, w2] := exec1[w1] and P2(w1, w2) ;
   if exec2[w1, w2] then incr(count) endif
endif ;
barrier ;
terminated := (count = 0) ;
barrier ;
return (exec2[w1, w2])
```

Figure 7.3: Function *executed* for two nested while loops

7.2.3 Correctness

Let us verify that a target loop program whose time loops are bounded with *terminated* does not terminate too early.

Lemma 23. *The implementation of terminated via the counters is correct.*

Proof. We prove this fact informally. The following properties ensure that, at a given time step t, *terminated* is not set to *tt* if some while loop iteration has not terminated in the execution domain:

- For every tooth in every dimension, *count* is incremented once (at its root) and decremented once (at its tip)—in this order. During execution every tooth contributes 1 to the global value of *count*, whereas before the start and after termination there is no contribution to *count*.

- Barrier synchronization ensures that all updates of *count* occur before the processors read the value of *count*. Note that the order in which increments and decrements take place does not affect the final value.
- If there is at least one processor evaluating some $executed_r(x_1, \cdots, x_d)$ $(1 \le r \le d)$ to tt at time t then the tooth τ at level r and through the point $(x_1, \cdots, x_r, lb_{r+1}, \cdots, lb_d)$ has started but not yet finished execution. Thus, at this point in time, τ is contributing 1 to *count*.
- Since τ contributes 1 to *count* and since there cannot have been more decrements than increments, *count* must be strictly positive, thus preventing termination.

7.2.4 Optimization

The straight-forward implementation of the counter scheme in Figure 7.2 has an essential drawback: there is only one shared counter which can be updated by any iteration, i.e., this counter is a bottleneck.

A better implementation would use multiple counters, each of which is only responsible for one r-dimensional subspace, thus avoiding many conflicts. As soon as such a counter becomes zero, the counter responsible for the next outer dimension is decremented. E.g., if we substitute r by d we get the scheme described before; if we substitute r by 1 we use one counter per tooth.

Note that in the latter case there may still be conflicting accesses of the counters: if all teeth terminate at the same time, then the teeth started by some tooth τ cause the counter of τ to be decremented, which terminates τ, and so on. All in all, we can have linearly many conflicting accesses of counters.

Another optimization is necessary for bounding the size of array *exec*, for which we gave no bound so far. J.-F. Collard [12] presents a way of determining a bound for arrays by calculating the life time of the array elements and then introducing reassignments.

7.2.5 The Example

Let us now apply the counter scheme in the development of a synchronous and, thus, unscannable target loop nest for our example program. For simplicity, Figures 7.4 to 7.6 show the target loop nest and some auxiliary functions before partitioning.

The target loop nest is presented in Figure 7.4, where $(c\ ?\ e_1\ :\ e_2)$ denotes a conditional expression whose value is e_1 if condition c evaluates to tt and e_2 otherwise. Note that, due to the imperfect nesting, we must use a separate predicate *terminated* (and, thus, a separate counter) for every source while dimension. In addition, we store the maximum value of all upper bounds of a loop at level l in max_index_l. For while loops the value of this variable is not valid before the corresponding while loop terminates, i.e., max_index_l contains a valid value when $terminated_l$ is tt.

Let us now consider the guards in more detail. In principle there are rather simple guards for non-loop statements and more complex guards for loop statements. We discuss the structure of the functions *executed* for the two cases by taking one example for each case; the guards for the other statements are very similar.

For a guard of a non-loop statement, we choose arbitrarily predicate *executed* of statement S_5 (Figure 7.5). The then branch of $executed\_S5$ checks for violations of the constraints of

```
for t1 := 0 while t1 ≤ (not terminated_1 ? t1 : (not terminated_2 ? t1 :
            max(4 * max_index_2+3+max_index_1, max_index_1))) do
   parfor p1 := min(t1−1, 0) to t1 do
      if executed_S1(t1, p1) then
         skip
      endif
      if executed_S2(t1, p1) then
         rt[t1−1, 0] := t1−1
      endif
      if executed_S3(t1, p1) then
         nxt[t1−1] := 1
      endif
      if executed_S4(t1, p1) then
         skip
      endif
      if executed_S5(t1, p1) then
         if_cond[p1, (t1−p1−5)/4] := (not Tag[p1, rt[p1, (t1−p1−5)/4]])
      endif
      if executed_S6(t1, p1) and if_cond[p1, (t1−p1−6)/4] then
         tag[p1, rt[p1, (t1−p1−6)/4]] := tt
      endif
      if executed_S7(t1, p1) and if_cond[p1, (t1−p1−6)/4] then
         skip
      endif
      if executed_S9(t1, p1) and if_cond[p1, (t1−p1−8)/4] then
         nxt[p1] := nxt[p1]+
            nrsuc[rt[p1, (t1−p1−8)/4]]
      endif
      for t2 := 0 to max_for_0_0_0 do
         if executed_S8(t1, p1, t2) and if_cond[p1, (t1−p1−7)/4] then
            rt[p1, t2+nxt[p1]] := suc[rt[p1, (t1−p1−7)/4], t2]
         endif
      enddo
   enddo
enddo
```

Figure 7.4: The synchronous target program

the target *index* space, whereas the else branch checks for the remaining index points whether the current point belongs to the target *execution* space.

As a representative of function *executed* of a loop statement we select *executed_S4* (Figure 7.6). Function *executed_S4* first computes the new value of *executed* at the current point. Then, it actualizes the counters and the variables *max_index* storing the maximal loop bounds of a dimension. Between the synchronizations via the barrier, *terminated* is computed and the value of *executed* is returned. For implementation reasons, the value of

function $executed\_S5(t1, p1)$: boolean
if $t1 < 5$ or $p1 < 0$ or $p1 > t1-5$ or $(t1-p1-5)\%4$ then
  return (ff)
else
  return ($exec_2[p1, (t1-p1-5)/4]$)
endif

Figure 7.5: $executed\_S5$

function $executed\_S4(t1, p1)$: boolean
  if $t1 \geq 4$ and $p1 \geq 0$ and $p1 \leq t1-4$ and $(t1-p1-4)\%4 = 0$ then
    if $(t1-p1-4)/4 = 0$ then
      $exec_2[p1, (t1-p1-4)/4] := (rt[p1, (t1-p1-4)/4] \neq \perp)$ and $exec_1[p1]$
    else
      $exec_2[p1, (t1-p1-4)/4] := (rt[p1, (t1-p1-4)/4] \neq \perp)$ and $exec_2[p1, ((t1-p1-4)/4) - 1]$
    endif
    if $exec_2[p1, (t1-p1-4)/4]$ then
      skip /* would be $incr(count_3)$ if there were an inner while loop */
    elseif $((t1-p1-4)/4 > 0\ ?\ exec_2[p1, (t1-p1-4)/4 - 1]\ :\ exec_1[p1])$ then
      $decr(count_2)$
      $max\_index_2 := max(max\_index_2, (t1-p1-4)/4)$
    endif
  else
    $local\_index\_violation\_flag := \mathit{tt}$
  endif
barrier
if $count_2 = 0$ then
  $terminated_2 := \mathit{tt}$
endif
barrier
if $local\_index\_violation\_flag$ then
  return (ff)
else
  return ($exec_2[p1, (t1-p1-4)/4]$)
endif

Figure 7.6: $executed\_S4$

executed at points outside of the index space is not stored in the array *exec* but in a local flag *local_index_violation_flag*. Note that the computed value of *executed* is stored in array *exec*, which allows us to access this value without re-calling function *executed*; this avoids the undesired re-computation of the side effects in *executed*.

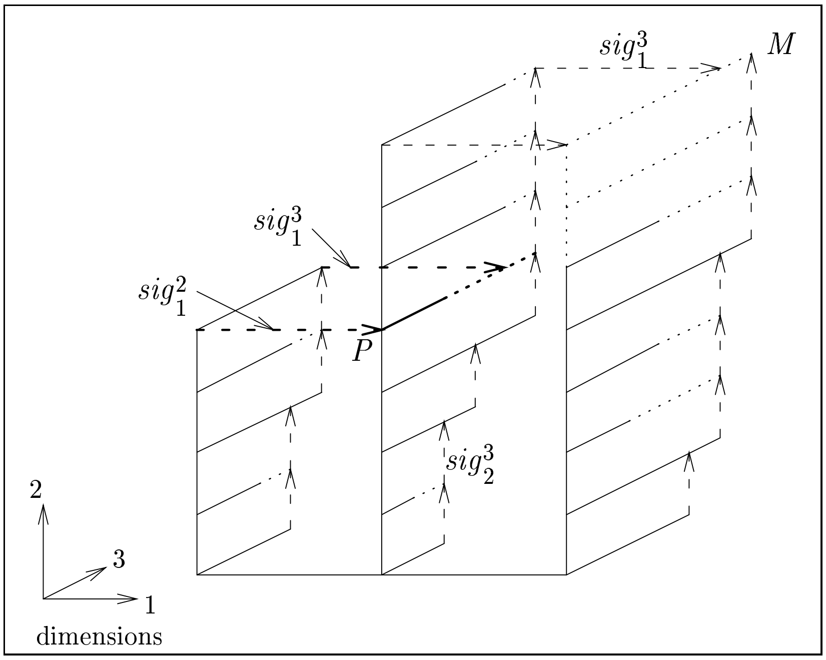

Figure 7.7: A three-dimensional comb

7.3 Termination Detection with Distributed Memory

In this section we present a solution of the termination problem that requires only local communication.

7.3.1 Idea

The basic idea of our solution is as follows: if (carefully selected) teeth along dimension r of the execution space inform their (still executing) neighbors in dimensions $1, \cdots, r-1$ of their termination, the maximal coordinates of every dimension of the execution space are communicated.

If we ensure that no tooth terminates before it has been informed of the termination of its neighbors, points that are involved in these communications are partially maximal until the point $(x_1, \cdots, x_d)$ in $\mathcal{S}$ is reached whose coordinates are all maximal, i.e., have the property $(\forall\, x' : x' \in \mathcal{X} : (\forall\, r : 1 \leq r \leq d : x'_r \leq x_r))$. When scanning this point, we can terminate all target loops.

The propagation of the maxima, up to level r, proceeds by valueless signals. Signal $sig^r_k(x_1, \cdots, x_r, lb_{r+1}, \cdots, lb_d)$ starts at point $(x_1, \cdots, x_r, lb_{r+1}, \cdots, lb_d)$ at level r and is sent to the neighboring tooth in direction k, where k is some outer level with respect to r, i.e., $k < r$.

The main problem is how to establish whether a tooth can terminate immediately when the corresponding while condition is violated or whether it has to wait for some signal first.

Example 10. Consider the three-dimensional comb of Figure 7.7.

Our scheme is more easily understood in the synchronous model.

In the figure, the teeth of $\mathcal{X}$ are represented by solid lines. Some teeth are extended by dotted lines, indicating that they are waiting for at least one signal. Points (of $\mathcal{S}$) on dotted lines do not execute the loop body, they only wait for signals. Signals are represented by dashed arrows.

Our aim is to identify the point M whose coordinates are maximal in every dimension. The first coordinate of M is quite easy to determine: it is the value at which the outermost loop terminates.

The second coordinate of M is the maximum of the lengths of all teeth pointing up (in the figure). To determine it, every vertical tooth tells its right neighbor the maximum of its own height and the maximal height left of it. This is the meaning of sig_1^2. If a vertical tooth is ready to terminate but did not yet receive sig_1^2 from its left neighbor, it must wait (except for the leftmost tooth) until this signal is received. (In the figure, the tooth at the right must wait.) Then it itself sends sig_1^2 on to its right neighbor and terminates. The following formal property holds for all teeth in dimension 2:

$$sig_1^2(x_1, x_2, lb_3) \Rightarrow (\forall\, x_1', x_2' \;:\; (x_1', x_2', lb_3') \in \mathcal{X} \wedge x_1' \leq x_1 \;:\; x_2' \leq x_2)$$

The determination of the maximal depth of teeth in each vertical plane (x_1 constant) proceeds analogously. Signals sig_2^3 are sent from every (perhaps extended, since waiting) tooth τ along dimension 3 to its upper neighbor of that plane, indicating that the current depth (the length of τ) is maximal for all teeth in dimension 3 to the left and including τ, for fixed x_1. Formally:

$$sig_2^3(x_1, x_2, x_3) \Rightarrow (\forall\, x_2', x_3' \;:\; (x_1, x_2', x_3') \in \mathcal{X} \wedge x_2' \leq x_2 \;:\; x_3' \leq x_3)$$

To combine the maxima of all vertical planes, the maximal point of each plane sends a signal sig_1^3 to its right neighbor. Again, it is important that this right neighbor must not terminate before the signal is received. Which teeth must wait? The maximal depth in every vertical plane is reached at the end of the (perhaps extended) vertical tooth that forms the base of this plane. This height was propagated to the right neighbor by sig_1^2. At that height, the maximal depth will also be propagated. Therefore, the tooth, rooted at that point (e.g., P in the figure) which received sig_1^2 and which points into dimension 3 (the thick tooth in the figure), must wait until sig_1^3 is received. Again, formally:

$$sig_1^3(x_1, x_2, x_3) \Rightarrow (\forall\, x_1', x_2', x_3' \;:\; (x_1', x_2', x_3') \in \mathcal{X} \wedge x_1' \leq x_1 \;:\; x_2' \leq x_2 \wedge x_3' \leq x_3)$$

The formal properties implied by the signals form a pattern that we call the partial maximality (of a point). M is partially maximal with respect to all dimensions and is, therefore, the maximal point.

7.3.2 Formalization

In the following, we define partial maximality recursively for an arbitrary number of dimensions. Then we construct a mechanism that sends signals from partially maximal points to the appropriate destinations.

To include the host of the processor array, we introduce a little hack. We imagine one more dimension, 0, which has extent 2. The polyhedron is located at position 0, and the host at position 1. Then we introduce signals that travel from position 0 to position 1. They are meant to communicate the termination of the target loops to the host.

Definition 24 (Partial maximality m_k^r). $m_k^r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)$ iff for fixed indices at levels 1 to $k-1$ and for points $(x_1',\cdots,x_r')$ below $(x_1',\cdots,x_r',lb_{r+1}',\cdots,lb_d')$ at level k $(x_k'\leq x_k)$, point $(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)$ is maximal in all dimensions $k+1,\cdots,r$. Formally:

$$
\begin{aligned}
&(\forall\, r\ :\ 0<r\leq d\ :\ (\forall\,(x_1,\cdots,x_r) : (x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)\in\widehat{\mathcal{X}}\ :\\
&\quad(\forall\, k\ :\ 0\leq k<r\ :\ m_k^r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)\ =\\
&\qquad(\forall\, x_k',\cdots,x_r'\ :\ (x_1,\cdots,x_{k-1},x_k',\cdots,x_r',lb_{r+1}',\cdots,lb_d')\in\widehat{\mathcal{X}}\wedge x_k'\leq x_k\ :\\
&\qquad\qquad x_{k+1}'\leq x_{k+1},\cdots,x_r'\leq x_r))))
\end{aligned}
$$

If $m_k^r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)$, we call point $(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)$ *partially maximal* with respect to dimensions $k+1$ to r.

Note that, for $k=0$, the right-hand side of Definition 24 simplifies to:

$$(\forall\, x_1',\cdots,x_r'\ :\ (x_1',\cdots,x_r',lb_{r+1}',\cdots,lb_d')\in\widehat{\mathcal{X}}\ :\ x_1'\leq x_1\wedge\cdots\wedge x_r'\leq x_r)$$

For our communication scheme of m by signals, we need an additional predicate, $w_k^r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)$, which indicates that the tooth that is rooted at $(x_1,\cdots,x_{r-1},lb_r,\cdots,lb_d)$ and extends along dimension r must wait until signal sig_k^r arrives at some point $(x_1,\cdots,x_{r-1},x_r',lb_{r+1},\cdots,lb_d)$ with $x_r'\geq x_r$. These additional points at which a tooth waits but executes nothing make the difference between $\mathcal{S}$ and $\mathcal{X}$. The lemmata that follow are valid for all points in $\mathcal{S}$.

Definition 25 (sig and w).

$$
\begin{aligned}
&(\forall\,(x_1,\cdots,x_d)\ :\ (x_1,\cdots,x_d)\in\mathcal{I}\ :\ (\forall\, k,r\ :\ 0\leq k<r\leq d\ :\ sig_k^r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)\ =\\
&\quad\neg executed_r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)\wedge\\
&\quad(\forall\, s\ :\ 1\leq s<r\ :\ \neg w_s^r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)\vee sig_s^r(x_1,\cdots,x_s-1,\cdots,x_r,lb_{r+1},\cdots,lb_d))\wedge\\
&\quad(r>k+1\Rightarrow sig_k^{r-1}(x_1,\cdots,x_{r-1},lb_r,\cdots,lb_d))\ ))
\end{aligned}
$$

For all other points sig is initialized with ff:

$$
\begin{aligned}
&(\forall\,(x_1,\cdots,x_d)\ :\ (x_1,\cdots,x_d)\in\mathbb{Z}^d-\mathcal{I}\ :\ (\forall\, k,r\ :\ 0\leq k<r\leq d\ :\\
&\quad sig_k^r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)\ =ff))
\end{aligned}
$$

$$
\begin{array}{llll}
\multicolumn{4}{l}{(\forall\,(x_1,\cdots,x_d)\ :\ (x_1,\cdots,x_d)\in\mathcal{I}\ :\ (\forall\, k,r\ :\ 0\leq k<r\leq d\ :\ w_k^r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)\ =}\\
\quad \mathsf{if} & k=0 & \rightarrow\ ff & (1)\\
\quad [] & r=k+1\wedge k>0\wedge x_{k+1}=lb_{k+1} & \rightarrow\ x_k\neq lb_k & (2)\\
\quad [] & r>k+1\wedge k>0\wedge x_{k+1}=lb_{k+1} & \rightarrow\ sig_k^{r-1}(x_1,\cdots,x_k-1,\cdots,x_{r-1},lb_r,\cdots,lb_d) & (3)\\
\quad [] & r\geq k+1\wedge k>0\wedge x_r>lb_r & \rightarrow\ w_k^r(x_1,\cdots,x_{r-1},x_r-1,lb_{r+1},\cdots,lb_d)\ \wedge & (4)\\
 & & \quad\neg sig_k^r(x_1,\cdots,x_k-1,\cdots,x_{r-1},x_r-1,lb_{r+1},\cdots,lb_d) & \\
\quad \mathsf{fi} &)) & &
\end{array}
$$

These equations can be explained as follows.

sig states that any point of a tooth that need not be executed and that does not have to wait for any signal sends signal sig_k^r if either the tooth and the signal lie in a two-dimensional plane (recursion base) or the root $(x_1,\cdots,x_{r-1},lb_r,\cdots,lb_d)$ of the tooth has already sent the signal into the same direction (recursion).

$w_k^r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)$ states whether the point $(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)$ at level r has to wait for some signal from direction k:

(1) No point has to wait for signals from the host.

(2) In every two-dimensional subspace (dimensions k and $r = k+1$), every tooth, i.e., at least the first point of it (with $x_{k+1} = lb_{k+1}$), has to wait for a signal from the tooth immediately preceding it—if any, i.e., if $x_k \neq lb_k$.

(3) In every at least three-dimensional subspace (dimensions k to $r > k+1$), every tooth parallel to dimension r, i.e., at least the first point of it, has to wait for some signal from direction k iff its root at level $r-1$ has received a signal from the same direction k.

(4) Any point of a tooth that is not the first point has to wait for a signal iff its predecessor on the tooth had to wait and did not receive the signal it was waiting for.

7.3.3 Signals and their Significance for Local Maximality

The main result of this subsection is that signals *sig* correctly propagate property m of local maximality. We state this in two separate lemmata.

Lemma 26 (Local maximum). *A point of some tooth along dimension* r *that need not be executed with respect to* r *and need not wait for a signal is maximal with respect to dimension* r*. Formally:*

$$\begin{array}{l}(\forall\ \ x_1,\cdots,x_r\ :\ (x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)\in\mathcal{S}\ :\ \neg executed_r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)\ \wedge\\ \quad (\forall\ s\ :\ 1\leq s<r\ :\ (\neg w_s^r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)\ \vee\\ \qquad\qquad sig_s^r(x_1,\cdots,x_s-1,\cdots,x_r,lb_{r+1},\cdots,lb_d)))\ \Rightarrow\\ \quad (\forall\ x_r'\ :\ (x_1,\cdots,x_{r-1},x_r',lb_{r+1},\cdots,lb_d)\in\mathcal{S}\ :\ x_r'\leq x_r))\end{array}$$

Proof. We prove the inverse implication: $(H \Rightarrow C) \Leftrightarrow (\neg C \Rightarrow \neg H)$.

$$\begin{array}{ll} & \neg(\forall\ x_r'\ :\ (x_1,\cdots,x_{r-1},x_r',lb_{r+1},\cdots,lb_d)\in\mathcal{S}\ :\ x_r'\leq x_r)\\ \Leftrightarrow & \{\text{ predicate calculus }\}\\ & (\exists\ x_r'\ :\ (x_1,\cdots,x_{r-1},x_r',lb_{r+1},\cdots,lb_d)\in\mathcal{S}\ :\ x_r'>x_r)\\ \Rightarrow & \{\text{ if point } (x_1,\cdots,x_{r-1},x_r',lb_{r+1},\cdots,lb_d) \text{ is scanned, it must be exe-}\\ & \quad\text{cuting or waiting }\}\\ & (\exists\ \ x_r'\ :\ (x_1,\cdots,x_{r-1},x_r',lb_{r+1},\cdots,lb_d)\in\mathcal{S}\ :\ executed_r(x_1,\cdots,x_{r-1},x_r',lb_{r+1},\cdots,lb_d)\ \vee\\ & \quad(\exists\ \ s\ :\ 1\leq s<r\ :\ (w_s^r(x_1,\cdots,x_{r-1},x_r',lb_{r+1},\cdots,lb_d)\wedge\\ & \quad\neg sig_s^r(x_1,\cdots,x_s-1,\cdots,x_r',lb_{r+1},\cdots,lb_d)))\\ \Rightarrow & \{\text{ predicate calculus }\}\\ & \neg(\forall\ \ (x_1,\cdots,x_r)\ :\ (x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)\in\mathcal{S}\ :\neg executed_r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)\ \wedge\\ & \quad(\forall\ s\ :\ 1\leq s<r\ :\ \neg w_s^r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)\vee sig_s^r(x_1,\cdots,x_s-1,\cdots,x_r,lb_{r+1},\cdots,lb_d)))\end{array}$$

In the following lemma, $\mathcal{SUB}_k^r(x_1,\cdots,x_{k-1})$ is the subspace of $\mathcal{S}$ in dimensions k to r $(1\leq k<r\leq d)$ and at fixed coordinates $(x_1,\cdots,x_{k-1})$.

Lemma 27 (*sig* implements m).

$$\begin{array}{l}(\forall\ k,r\ :\ 0\leq k<r\leq d\ :\ (\forall\ (x_1,\cdots,x_r)\ :\ (x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)\in\mathcal{S}\ :\\ \quad sig_k^r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)\Rightarrow m_k^r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)))\end{array}$$

Proof. By induction on the dimension of $\mathcal{SUB}_k^r(x_1, \cdots, x_{k-1})$.

- Induction base ($r = k+1$):
  For $sig_k^{k+1}(x_1, \cdots, x_{k-1}, lb_k, \cdots, lb_d) \Rightarrow m_k^{k+1}(x_1, \cdots, x_{k-1}, lb_k, \cdots, lb_d)$ we must distinguish two cases, since x_0 does not exist.
  - First case: $k = 0$

    $$\begin{array}{ll} & sig_0^1(x_1, lb_2, \cdots, lb_d) \\ \Leftrightarrow & \{ \text{ definition of } sig \text{ with } k = 0,\ r = 1 \ \} \\ & \neg executed_1(x_1, lb_2, \cdots, lb_d) \\ \Rightarrow & \{ \text{ Lemma 26 with } k = 0,\ r = 1 \ \} \\ & (\forall\, x'_1 \ :\ (x'_1, lb_2, \cdots, lb_d) \in \mathcal{S} \ :\ x'_1 \leq x_1) \\ \Leftrightarrow & \{ \text{ definition of } m_0^1 \ \} \\ & m_0^1(x_1, lb_2, \cdots, lb_d) \end{array}$$

  - Second case: $k > 0$
    We prove $sig_k^{k+1}(x_1, \cdots, x_{k-1}, lb_k, \cdots, lb_d) \Rightarrow m_k^{k+1}(x_1, \cdots, x_{k-1}, lb_k, \cdots, lb_d)$ by induction on x_k.
    - Induction base ($x_k = lb_k$):

      $$\begin{array}{ll} & sig_k^{k+1}(x_1, \cdots, x_{k-1}, lb_k, \cdots, lb_d) \\ \Leftrightarrow & \{ \text{ definition of } sig \text{ with } x_k = lb_k \text{ and simplification for } r = k+1 \ \} \\ & \neg executed_{k+1}(x_1, \cdots, x_{k-1}, lb_k, \cdots, lb_d) \ \wedge (\forall\, s \ :\ 1 \leq s < k+1 \ : \\ & \quad \neg w_s^{k+1}(x_1, \cdots, x_{k-1}, lb_k, \cdots, lb_d) \vee \\ & \quad sig_s^{k+1}(x_1, \cdots, x_s - 1, \cdots, x_{k+1}, lb_{k+2}, \cdots, lb_d)) \\ \Rightarrow & \{ \text{ Lemma 26 } \} \\ & (\forall\, x'_{k+1} \ :\ (x_1, \cdots, x_k, x'_{k+1}, lb_{k+2}, \cdots, lb_d) \in \mathcal{S} \ :\ x'_{k+1} \leq x_{k+1}) \\ \Leftrightarrow & \{ \ lb_k \text{ is the smallest } x_k\text{; introduction of a new dummy with empty} \\ & \quad \text{range } \} \\ & (\forall\, x'_k, x'_{k+1} \ :\ (x_1, \cdots, x_{k-1}, x'_k, x'_{k+1}, lb_{k+2}, \cdots, lb_d) \in \mathcal{S} \ \wedge \ x'_k \leq x_k \ : \\ & \qquad x'_{k+1} \leq x_{k+1}) \\ \Leftrightarrow & \{ \text{ definition of } m \ \} \\ & m_k^{k+1}(x_1, \cdots, x_{k-1}, lb_k, \cdots, lb_d) \end{array}$$

    - Induction step ($x_k - 1 \rightarrow x_k$, where $x_k > lb_k$):

      $$\begin{array}{ll} & sig_k^{k+1}(x_1, \cdots, x_{k-1}, lb_k, \cdots, lb_d) \\ \Leftrightarrow & \{ \text{ definition of } sig \text{ with } x_k > lb_k \text{ and simplification for } r = k+1 \ \} \\ & \neg executed_{k+1}(x_1, \cdots, x_{k-1}, lb_k, \cdots, lb_d) \ \wedge (\forall\, s \ :\ 1 \leq s < k+1 \ : \\ & \quad (\neg w_s^{k+1}(x_1, \cdots, x_{k-1}, lb_k, \cdots, lb_d) \vee \\ & \quad sig_s^{k+1}(x_1, \cdots, x_s - 1, \cdots, x_{k+1}, lb_{k+2}, \cdots, lb_d))) \\ \Leftrightarrow & \{ \text{ duplicate of} \\ & \quad \neg w_k^{k+1} \vee sig_k^{k+1}(x_1, \cdots, x_{k-1}, x_k - 1, x_{k+1}, lb_{k+2}, \cdots, lb_d) \ \} \end{array}$$

$$
\begin{array}{ll}
& \neg executed_{k+1}(x_1,\cdots,x_{k-1},lb_k,\cdots,lb_d) \ \wedge (\forall\, s \,:\, 1 \le s < k+1 \,: \\
& \quad (\neg w_s^{k+1}(x_1,\cdots,x_{k-1},lb_k,\cdots,lb_d) \vee \\
& \quad sig_s^{k+1}(x_1,\cdots,x_s-1,\cdots,x_{k+1},lb_{k+2},\cdots,lb_d))) \ \wedge \\
& \quad \neg w_k^{k+1}(x_1,\cdots,x_{k-1},lb_k,\cdots,lb_d) \vee \\
& \quad sig_k^{k+1}(x_1,\cdots,x_{k-1},x_k-1,x_{k+1},lb_{k+2},\cdots,lb_d) \\
\Rightarrow & \quad \{ \text{ Lemma 26 } \} \\
& (\forall\, x'_{k+1} \,:\, (x_1,\cdots,x_k,x'_{k+1},lb_{k+2},\cdots,lb_d) \in \mathcal{S} \,:\, x'_{k+1} \le x_{k+1}) \ \wedge \\
& \quad \neg w_k^{k+1}(x_1,\cdots,x_{k-1},lb_k,\cdots,lb_d) \vee \\
& \quad sig_k^{k+1}(x_1,\cdots,x_{k-1},x_k-1,x_{k+1},lb_{k+2},\cdots,lb_d) \\
\Rightarrow & \quad \{ \text{ by structural induction on the definition of } w \ \} \\
& (\forall\, x'_{k+1} \,:\, (x_1,\cdots,x_k,x'_{k+1},lb_{k+2},\cdots,lb_d) \in \mathcal{S} \,:\, x'_{k+1} \le x_{k+1}) \ \wedge \ (x_k = lb_k \ \vee \\
& \quad (\exists\, \widehat{x_{k+1}} \,:\, \widehat{x_{k+1}} \le x_{k+1} \,:\, sig_k^{k+1}(x_1,\cdots,x_k-1,\widehat{x_{k+1}},lb_{k+2},\cdots,lb_d))) \\
\Leftrightarrow & \quad \{ \ x_k \ne lb_k \text{ (induction step) } \} \\
& (\forall\, x'_{k+1} \,:\, (x_1,\cdots,x_k,x'_{k+1},lb_{k+2},\cdots,lb_d) \in \mathcal{S} \,:\, x'_{k+1} \le x_{k+1}) \ \wedge \\
& \quad (\exists\, \widehat{x_{k+1}} \,:\, \widehat{x_{k+1}} \le x_{k+1} \,:\, sig_k^{k+1}(x_1,\cdots,x_k-1,\widehat{x_{k+1}},lb_{k+2},\cdots,lb_d)) \\
\Rightarrow & \quad \{ \text{ induction hypothesis for } sig_k^{k+1}(x_1,\cdots,x_k-1,\widehat{x_{k+1}},lb_{k+2},\cdots,lb_d) \ \} \\
& (\forall\, x'_{k+1} \,:\, (x_1,\cdots,x_k,x'_{k+1},lb_{k+2},\cdots,lb_d) \in \mathcal{S} \,:\, x'_{k+1} \le x_{k+1}) \ \wedge \\
& \quad (\exists\, \widehat{x_{k+1}} \,:\, \widehat{x_{k+1}} \le x_{k+1} \,:\, m_k^{k+1}(x_1,\cdots,x_k-1,\widehat{x_{k+1}},lb_{k+2},\cdots,lb_d)) \\
\Leftrightarrow & \quad \{ \ \widehat{x_{k+1}} \le x_{k+1}; \text{ definition of } m) \ \} \\
& (\forall\, x'_{k+1} \,:\, (x_1,\cdots,x_k,x'_{k+1},lb_{k+2},\cdots,lb_d) \in \mathcal{S} \,:\, x'_{k+1} \le x_{k+1}) \ \wedge \\
& \quad m_k^{k+1}(x_1,\cdots,x_k-1,x_{k+1},lb_{k+2},\cdots,lb_d) \\
\Leftrightarrow & \quad \{ \text{ definition of } m \ \} \\
& (\forall\, x'_{k+1} \,:\, (x_1,\cdots,x_k,x'_{k+1},lb_{k+2},\cdots,lb_d) \in \mathcal{S} \,:\, x'_{k+1} \le x_{k+1}) \ \wedge (\forall\, x'_k, x'_{k+1} \,: \\
& \quad (x_1,\cdots,x_{k-1},x'_k,x'_{k+1},lb_{k+2},\cdots,lb_d) \in \mathcal{S} \ \wedge \ x'_k \le x_k-1 \,:\, x'_{k+1} \le x_{k+1}) \\
\Leftrightarrow & \quad \{ \text{ combination of two quantification ranges } \} \\
& (\forall x'_k, x'_{k+1} \,:\, (x_1,\cdots,x_{k-1},x'_k,x'_{k+1},lb_{k+2},\cdots,lb_d) \in \mathcal{S} \ \wedge \\
& \quad x'_k \le x_k \,:\, x'_{k+1} \le x_{k+1}) \\
\Leftrightarrow & \quad \{ \text{ definition of } m \ \} \\
& m_k^{k+1}(x_1,\cdots,x_{k-1},lb_k,\cdots,lb_d)
\end{array}
$$

- Induction step ($k \to k-1$, where $r-k > 1$):

$$
\begin{array}{ll}
& sig_{k-1}^r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d) \\
\Leftrightarrow & \quad \{ \text{ definition of } sig \text{ with } r > k+1 \ \} \\
& \neg executed_r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d) \ \wedge \ (\forall\, s \,:\, 1 \le s < r \,: \\
& \quad \neg_s^r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d) \vee sig_s^r(x_1,\cdots,x_s-1,\cdots,x_r,lb_{r+1},\cdots,lb_d)) \ \wedge \\
& \quad sig_{k-1}^{r-1}(x_1,\cdots,x_{r-1},lb_r,\cdots,lb_d) \\
\Rightarrow & \quad \{ \text{ Lemma 26 } \} \\
& (\forall\, x'_r \,:\, (x_1,\cdots,x_{r-1},x'_r,lb_{r+1},\cdots,lb_d) \in \mathcal{S} \,:\, x'_r \le x_r) \ \wedge \\
& \quad sig_{k-1}^{r-1}(x_1,\cdots,x_{r-1},lb_r,\cdots,lb_d) \\
\Rightarrow & \quad \{ \text{ induction hypothesis for } sig_{k-1}^{r-1} \ \} \\
& (\forall\, x'_r \,:\, (x_1,\cdots,x_{r-1},x'_r,lb_{r+1},\cdots,lb_d) \in \mathcal{S} \,:\, x'_r \le x_r) \ \wedge \\
& \quad m_{k-1}^{r-1}(x_1,\cdots,x_{r-1},lb_r,\cdots,lb_d) \\
\Leftrightarrow & \quad \{ \text{ definition of } m \ \}
\end{array}
$$

$$
\begin{array}{ll}
& (\forall\, x'_r \,:\, (x_1,\cdots,x_{r-1},x'_r,lb_{r+1},\cdots,lb_d)\in\mathcal{S} \,:\, x'_r \le x_r) \;\wedge \\
& \quad (\forall\, x'_{k-1},\cdots,x'_{r-1} \,:\, (x_1,\cdots,x_{k-2},x'_{k-1},\cdots,x'_{r-1},lb_r,\cdots,lb_d)\in\mathcal{S} \;\wedge\; x'_{k-1}\le x_{k-1} \,: \\
& \quad x'_k \le x_k \wedge \cdots \wedge x'_{r-1}\le x_{r-1}) \\
\Rightarrow & \quad \{ \text{ combination of two quantification ranges } \} \\
& (\forall\, x'_k,\cdots,x'_r \,:\, (x_1,\cdots,x_{k-2},x'_{k-1},\cdots,x'_r,lb_{r+1},\cdots,lb_d)\in\mathcal{S} \;\wedge\; x'_{k-1}\le x_{k-1} \,: \\
& \quad x'_k \le x_k \wedge \cdots \wedge x'_r \le x_r) \\
\Leftrightarrow & \quad \{ \text{ definition of } m \text{ } \} \\
& m^r_{k-1}(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)
\end{array}
$$

Our aim has been to identify the point of $\mathcal{S}$ with maximal coordinates in all dimensions. The scanning of this point indicates the termination of the entire target loop nest. We have constructed a signaling scheme in which this point sends signal sig^d_0 to the host.

Remark 28 (Optimization).

- A tooth in direction r must send at least one signal, into direction $r-1$.
- One simple optimization can be made immediately: signals need not be sent to points x along a tooth that has terminated, i.e., points that neither are active nor wait for a signal. Thus, $sig^r_k(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)$ implies that, for any k' with $k'<k$, $sig^r_{k'}(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)$ need not be sent. In Example 10 we have already omitted these signals, but they are of course part of Definition 25. Thus every terminating tooth has to send at most one signal.
- In summary, every tooth must send exactly one signal when it terminates.

7.3.4 Target Code Generation for Distributed Memory Machines

In the remainder of this section we derive code for the target program. In a straight-forward implementation we could augment the source program by an implementation of the predicates and the signals, developed in the previous sections, and apply the space-time mapping to this augmented source program.

However, since the transformation of the signaling scheme is problem-independent we decided to derive a skeleton for the transformed signaling scheme which can be filled with the problem-specific body statements and transformations. This strategy also unburdens the target generator of a parallelizing compiler significantly, which accelerates the target code generation.

Thus, we present an augmentation of the *target* loop body such that the irregular shape of the transformed execution space is dealt with properly. Our augmentation implements the signaling scheme presented above. We prove that, while the target loops enumerate $\mathcal{TS}$, the augmented body reduces the execution of the loop nest to precisely the points of $\mathcal{TX}$.

7.3.4.1 General Technique

First we present the target code which is abstract in the sense that it is neither optimized for memory usage (it is single-assignment!) nor adapted to the execution model (synchronous or asynchronous) of the target machine. These adaptations are given in Section 7.3.4.3.

The target code must specify communications (send and receive primitives) explicitly. We use three primitives that are executable on the PARIX[1] operating system [45], one for transmitting and two for receiving. The reception mode depends on the type of message transmitted.

Our signaling scheme contains two types of messages.

- One type is responsible for the propagation of information along one tooth. An example is the value of the predicate *active*. The receipt of a message of this type is necessary for the execution of the loop body at the respective point.
- The other type of messages is for signals *sig*. These signals must be "probed" [42], since execution of a loop iteration may proceed without their receipt.

Let us briefly describe the three primitives:

- asend(*dir*, *list_of_vals*) is a PARIX command that transmits a list of values into a specified direction in $(d-1)$-dimensional space. Communication is asynchronous to prevent deadlocks.
- receive(*dir*, *list_of_vals*) is a PARIX command that performs a blocking receive of a list of values from a specified direction.
- creceive(*dir*, *list_of_vals*) is a command defined by us that performs a non-blocking receive of a list of values from a specified direction. Martin [42] defines the value of a probe $\overline{Y}$ on some communication action Y as a boolean value that indicates whether the corresponding communication action is pending. In our context, Y is a receive command from some direction and, thus, $\overline{Y}$ holds iff the corresponding asend command did already take place. With this construct, we define creceive(*dir*, *list_of_vals*) as follows:

  $$\textsf{if}\ \overline{\textsf{receive}(dir, list\_of\_vals)}\ \textsf{then}\ \ \textsf{receive}(dir,\ list\_of\_vals)\ \ \textsf{endif}.$$

  Note: $\overline{\textsf{receive}(dir, list\_of\_vals)}$ corresponds roughly to the PARIX command Select(ReceiveOption(*dir*)) [45].

We are now able to present the single-assignment target program. For readability we only describe the case of a perfect loop nest; the technical modifications for the general case can be found in [33]. Also for simplicity, we implement termination by the PARIX command AbortServer which is executed only by the iteration that is maximal for all dimensions (i.e., partially maximal with respect to dimensions 1 to d, as defined in Definition 24). Thus, the while loops in the target program need no upper bound.

The skeleton of the target loop nest is displayed in Figure 7.8 and refined in the subsequent Figures 7.9 and 7.10. This code is for asynchronous parallel execution on machines with distributed memory. The necessary changes for synchronous execution and/or shared memory are discussed in Section 7.3.4.3.

Note that the loops in the figure represent the worst case of a nest with only while loops, which are therefore sequential (remember that there is no parallel while loop). However, this is not typical, since the parallel loops (which hopefully do exist) can be written as (parallel) for loops (Remark 21 on page 49). Thus, some of the while loops in the scheme are replaced by parfor loops.

[1] PARIX^{TM} is an operating system for parallel computers with distributed memory by the PARSYTEC company based on the SPMD programming model [46].

```
k_1(lb_1) := 0
prg-active_1(lb_1) := tt
for y_1 := lb_1 while tt do
⋮
   for y_d := lb_d while tt do
      b_1^pre(x_1)
      ⋮
      b_d^pre(x_1, ..., x_d)
      if prg-active_d(x_1, ..., x_d) and condition_d(x_1, ..., x_d) then
         b(x_1, ..., x_d)
      endif
      b_1^post(x_1)
      ⋮
      b_d^post(x_1, ..., x_d)
   enddo
⋮
enddo
```

Figure 7.8: The target loop nest

The augmentation of the loop body for every level r, i.e., $b_r(x_1, \cdots, x_r)$, consists of two parts:

- The prefix $b_r^{\mathrm{pre}}(x_1, \cdots, x_r)$, executed before the transformed source loop body, is displayed in Figure 7.9. It receives all necessary data and calculates the output values of all variables that are introduced by our signaling scheme.
- The postfix $b_r^{\mathrm{post}}(x_1, \cdots, x_r)$, executed after the source loop (at least for asynchronous execution, see Remark 29), is displayed in Figure 7.10. It is responsible for sending all necessary information.

Let us now discuss the code in detail:

Variables:

- $chan_k^{\mathrm{S}}$ represents the channel for the signals that travel along dimension k. $chan_r^{\mathrm{D}}$ represents the channel for the data, i.e., $prg\text{-}active_r$, k_r, and $prg\text{-}w_k^r$, $1 \leq k < r$, that are propagated along every tooth.
- k_r is the direction in which the tooth must send a signal before it terminates. It corresponds to the lower index k of $sig_k^r(x_1, \cdots, x_r)$. The optimization outlined in Remark 28 ensures that there is a unique k for each tooth.
- $sigval_r$ corresponds to sig_k^r in Definition 25 (more precisely, it guarantees the first two conjuncts of Definition 25, cf. Lemma 35).

$(\forall\ r\ :\ 1 \leq r \leq d\ :\ b_r^{\text{pre}}(x_1, \cdots, x_r)\ :$

if $x_{r+1} = lb_{r+1}$ and $\cdots$ and $x_d = lb_d$ then

$A :$
/* *receive signals* */
creceive$(chan_1^{\text{S}}, [rcved_1^r(x_1, \cdots, x_r)])$
$\vdots$
creceive$(chan_{r-1}^{\text{S}}, [rcved_{r-1}^r(x_1, \cdots, x_r)])$

$B :$
/* *receive data* */
receive$(chan_r^{\text{D}}, [\mathit{prg\text{-}active}_r(x_1, \cdots, x_r),\ k_r(x_1, \cdots, x_r),$
$\mathit{prg\text{-}w}_1^r(x_1, \cdots, x_r), \cdots, \mathit{prg\text{-}w}_{r-1}^r(x_1, \cdots, x_r)])$

/* *calculate output values for all signals* */

$C :$
$\mathit{prg\text{-}w}_1^r(x_1, \cdots, x_r{+}1) := \mathit{prg\text{-}w}_1^r(x_1, \cdots, x_r)$ and not $rcved_1^r$
$\vdots$
$\mathit{prg\text{-}w}_{r-1}^r(x_1, \cdots, x_r{+}1) := \mathit{prg\text{-}w}_{r-1}^r(x_1, \cdots, x_r)$ and not $rcved_{r-1}^r$

$D :$
$\mathit{prg\text{-}w}_1^{r+1}(x_1, \cdots, x_r, lb_{r+1}) := rcved_1^r(x_1, \cdots, x_r)$
$\vdots$
$\mathit{prg\text{-}w}_{r-1}^{r+1}(x_1, \cdots, x_r, lb_{r+1}) := rcved_{r-1}^r(x_1, \cdots, x_r)$
$\mathit{prg\text{-}w}_r^{r+1}(x_1, \cdots, x_r, lb_{r+1}) := (x_r \neq lb_r)$
$\leftarrow$ only for $r < d$

$E :$ $k_r(x_1, \cdots, x_r + 1) := k_r(x_1, \cdots, x_r)$

if $\mathit{prg\text{-}active}_r(x_1, \cdots, x_r)$ and $\mathit{condition}_r(x_1, \cdots, x_r)$ then
$\mathit{sigval}_r(x_1, \cdots, x_r) := \mathit{ff}$
$\mathit{prg\text{-}active}_r(x_1, \cdots, x_r{+}1) := \mathit{tt}$

$F :$
$\mathit{prg\text{-}active}_{r+1}(x_1, \cdots, x_r, lb_{r+1}) := \mathit{tt}$
$k_{r+1}(x_1, \cdots, x_r, lb_{r+1}) := r$
$\leftarrow$ only for $r < d$

else

$G :$ $\mathit{prg\text{-}w}^r(x_1, \cdots, x_r) := \mathit{prg\text{-}w}_1^r(x_1, \cdots, x_r{+}1)$ or $\cdots$ or $\mathit{prg\text{-}w}_{r-1}^r(x_1, \cdots, x_r{+}1)$
$\mathit{sigval}_r(x_1, \cdots, x_r) :=$ not $\mathit{prg\text{-}w}^r(x_1, \cdots, x_r)$
$\mathit{prg\text{-}active}_r(x_1, \cdots, x_r{+}1) := \mathit{ff}$

$H :$
$\mathit{prg\text{-}active}_{r+1}(x_1, \cdots, x_r, lb_{r+1}) := \mathit{ff}$
if $\mathit{sigval}_r(x_1, \cdots, x_r)$ then
$k_{r+1}(x_1, \cdots, x_r, lb_{r+1}) := k_r(x_1, \cdots, x_r)$
else
$k_{r+1}(x_1, \cdots, x_r, lb_{r+1}) := r$
endif
$\leftarrow$ only for $r < d$

endif

endif

$)$

Figure 7.9: The prefix of the transformed loop body

$(\forall\, r : 1 \le r \le d : b_r^{\text{post}}(x_1, \cdots, x_r) :$

    if $x_{r+1} = lb_{r+1}$ and $\cdots$ and $x_d = lb_d$ then
$I:$
      if $k_r(x_1, \cdots, x_r) = 0$ and $sigval_r(x_1, \cdots, x_r)$ then
        Abort Server() $\leftarrow$ only for $r = d$
      endif
$J:$
      /* *send data* */
      asend($chan_r^{\mathrm{D}}$, [$prg\text{-}active_r(x_1, \cdots, x_r{+}1), k_r(x_1, \cdots, x_r{+}1),$
        $prg\text{-}w_1^r(x_1, \cdots, x_r{+}1), \cdots, prg\text{-}w_{r-1}^r(x_1, \cdots, x_r{+}1)$])
$K:$
      /* *send signals* */
      if $k_r(x_1, \cdots, x_r) \neq 0$ and $sigval_r(x_1, \cdots, x_r)$ then
        asend($chan_{k_r}^{\mathrm{S}}$, [tt])
      endif
    endif
)

Figure 7.10: The postfix of the transformed loop body

- $prg\text{-}active_r(x)$ is the counterpart of the predicate *active* in the source loop nest, i.e., for every point x of $\mathcal{I}$, the value of $prg\text{-}active_r(x)$ at the end of the program is equal to the value of $active_r(x)$. We say *prg-active implements* predicate *active* and prove this fact in Lemma 33. The value of $prg\text{-}active(x)$ is undefined if $\mathcal{T}x$ is not scanned by the target loop nest.
- Analogously, $prg\text{-}w_k^r$ is the counterpart of w_k^r in Definition 25.

Execution:

- The outermost if clause prevents the body from receiving, calculating and sending signals and messages that are not specified by our signaling scheme of Definition 25, i.e., signals at depth $r' > r$.
- Part A probes signals that are expected and receives those that are actually being sent. Thus, $rcved_k^r(x_1, \cdots, x_r)$ is equal to $sig_k^r(x_1, \cdots, x_k{-}1, \cdots, x_r)$.
- Part B propagates all necessary information along the tooth in direction r.
- Part C implements alternative (4) of Definition 25.
- All but the last line of part D implements alternative (3); the last line implements alternative (2).
- Line E copies the value of k for the next iteration.
- The if clause after line E tests whether the current iteration must be executed with respect to level r (Definition 16). The value of $prg\text{-}active_r$, $sigval_r$ and, if they exist, $prg\text{-}active_{r+1}$ and k_{r+1} depend on the outcome of the test.
- sig_k^r has three conjuncts (Definition 25). The first corresponds to the then or the else branch of said if clause, the second to the calculation at line G. The third conjunct is satisfied by an appropriate setting of k_r in parts F and H.

- *prg-active* implements *active* (Definition 15), as proved in Lemma 33.
- Part I applies only for $r=d$. It tests for sig_0^d (i.e., $sigval_d$ and $k_d=0$) and, if so, terminates the entire program (compare Lemma 27 and Definition 24).
- Part J sends the data that are received by Part B on to the next point of the tooth.
- Part K sends sig_k^r, if it has to be sent, i.e., if *sigval* holds. (At present, we ignore signals in direction 0, but one could probably use this information to develop smarter loop bounds.)

Remark 29. In the synchronous model, we obtain the same semantics if the postfixes are made prefixes instead. This is an optimization if each processor has a co-processor for the transmission of messages so that computation and message handling can proceed in parallel, as is the case for the transputer [34]. For asynchronous machines, there is no similar optimization since only the receipt of messages can guarantee that the sender has updated all transformed source variables.

7.3.4.2 Correctness Proof

In this section, we prove that the target program executes the transformed source loop body for all points whose inverse image is in the execution space.

For the following proofs, we need a formal definition of $\mathcal{TS}$, the set of target points that are scanned by the target program. Note that the target while loops have no upper bounds: they enumerate an infinite set. Our way of terminating the target program is by calling the PARIX command AbortServer at some point. This call terminates the whole program. Thus, points of $\mathcal{TI}$ are scanned until the AbortServer command is issued at some point. This leads to the following definition:

Definition 30 (*prg-scanned*, $\mathcal{S}$ and $\mathcal{TS}$). The image of a point $x\in\mathcal{I}$ is scanned by the target program if all points with a schedule not larger than $t(x)$ do not call the AbortServer command, where the call of AbortServer is guarded by the condition $k_r(x')=0 \wedge sigval_r(x')$ (Figure 7.10, Part I). Formally:

$$(\forall\ x\ :\ x\in\mathcal{I}\ :\ prg\text{-}scanned(x) = \\ (\forall\ x'\ :\ x'\in\mathcal{I}\wedge t(x')\leq t(x)\ :\ \neg(k_r(x')=0\wedge sigval_d(x'))))$$

$$\mathcal{S} = \{x\in\mathcal{I} : prg\text{-}scanned(x)\}$$

$$\mathcal{TS} = \{\mathcal{T}x\in\mathcal{TI} : prg\text{-}scanned(x)\}$$

To be able to reason about target points whose transformed loop body is executed, we need also a formal definition of those points. The target program executes the transformed loop body iff the point is scanned and $prg\text{-}active_d(x_1,\cdots,x_d)\wedge condition_d(x_1,\cdots,x_d)$ (Figure 7.8). Thus, we define a predicate *prg-exec* accordingly and prove (Theorem 41) that the transformed loop body is executed exactly for those points that belong to the execution space, i.e., that $prg\text{-}exec(y)$ is equal to $executed(\mathcal{T}^{-1}y)$ for all points $y\in\mathcal{TI}$.

Definition 31 (*prg-exec*).

$$(\forall\ x\ :\ x\in\mathcal{I}\ :\ prg\text{-}exec(x) = prg\text{-}scanned(x)\wedge prg\text{-}active_d(x)\wedge condition_d(x))$$

This definition makes sense only if *prg-active* is never reassigned. The following auxiliary lemma to that effect is proved informally.

Lemma 32. *Every variable* $\textit{prg-active}_r(x_1,\cdots,x_r)$*,* $1 \le r \le d$ *and* $(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d) \in \mathcal{I}$*, is assigned at most once during the execution of the transformed code.*

Proof. Every $\textit{prg-active}_r(x_1,\cdots,x_r)$ occurs exactly once as left hand side of an assignment. Three disjoint cases cover every point $(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)$ of $\mathcal{I}$:

1. if $(r=1 \wedge x_r=lb_r)$, it is assigned by the initialization statement $\textit{prg-active}_1(lb_1) := \textit{tt}$;
2. if $(1<r<d \wedge x_r=lb_r)$, it is assigned by $\textit{prg-active}_{r+1}(x_1,\cdots,x_r,lb_{r+1}) := \textit{tt}$ in the `then` branch of $b^{\text{pre}}_{r-1}(x_1,\cdots,x_{r-1})$;
3. if $x_r > lb_r$, it is assigned either by $\textit{prg-active}_r(x_1,\cdots,x_r+1) := \textit{tt}$ or by $\textit{prg-active}_r(x_1,\cdots,x_r+1) := \textit{ff}$ in the `then` branch or the `else` branch of $b^{\text{pre}}_r(x_1,\cdots,x_r)$, respectively.

Since no point is scanned more than once, we conclude that $\textit{prg-active}_r(x_1,\cdots,x_r)$ is assigned at most once for any $(x_1,\cdots,x_r)$.

In the succeeding lemmata we prove properties of the target program. Thus, we need to refer to values of program variables. In our single-assignment setting, we are only interested in the values *at the end of the execution of the target program.* This allows us to compute the values of some variables from the values of other variables by straight-forward code inspection.

First, we prove that predicate *active* is implemented correctly.

Lemma 33. $\textit{prg-active}_r$ *implements* $\textit{active}_r$ *for all points of* $\mathcal{I}$ *where the value of* $\textit{prg-active}_r$ *is defined, i.e., whose image is scanned by the target program. Formally:*

$$\begin{gathered}(\forall\, r,(x_1,\cdots,x_d)\;:\;1\le r\le d \wedge \mathcal{T}(x_1,\cdots,x_d)\in\mathcal{TS}\;:\\ \textit{prg-active}_r(x_1,\cdots,x_r) = \textit{active}_r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d))\end{gathered}$$

Proof. Induction over the nesting depth r, and then induction over the index range of the r loop.

- Induction base ($r=1$):
  - Induction base ($x_1 = lb_1$):

    $$\begin{array}{ll} & x_1 = lb_1 \\ \Rightarrow & \{\text{ definition of } \textit{prg-active} \text{ (Figure 7.8) and } \textit{active} \ \} \\ & \textit{prg-active}_1(x_1) = \textit{tt} = \textit{active}_1(x_1,lb_2,\cdots,lb_d) \end{array}$$

  - Induction step ($x_1-1 \to x_1$, where $x_1 > lb_1$):

    $$\begin{array}{ll} & \textit{prg-active}_1(x_1) \\ \Leftrightarrow & \{\text{ definition of } \textit{prg-active}_1(x_1) \text{ in the } \texttt{if} \text{ clause of } b_1(x_1-1)\text{: } \} \\ & \textit{prg-active}_1(x_1-1) \;\wedge\; \textit{condition}_1(x_1-1) \\ \Leftrightarrow & \{\text{ induction hypothesis for } x_1-1 \ \} \\ & \textit{active}_1(x_1-1,lb_2,\cdots,lb_d) \;\wedge\; \textit{condition}_1(x_1-1) \\ \Leftrightarrow & \{\text{ definition of } \textit{active}_r \text{ for } r=1 \text{ and } x_1>lb_1 \ \} \\ & \textit{active}_1(x_1,lb_2,\cdots,lb_d) \end{array}$$

- Induction step ($r-1 \rightarrow r$, where $r>1$):
  - Induction base ($x_r = lb_r$):

$$
\begin{array}{ll}
& \textit{prg-active}_r(x_1,\cdots,x_{r-1},lb_r) \\
\Leftrightarrow & \{\ \text{definition of } \textit{prg-active}_1(x_1) \text{ in the } \mathsf{if} \text{ clause of } b_{r-1}(x_1,\cdots,x_{r-1}):\ \} \\
& \textit{prg-active}_{r-1}(x_1,\cdots,x_{r-1})\ \wedge\ \textit{condition}_{r-1}(x_1,\cdots,x_{r-1}) \\
\Leftrightarrow & \{\ \text{induction hypothesis for } r-1\ \} \\
& \textit{active}_{r-1}(x_1,\cdots,x_{r-1},lb_r,\cdots,lb_d)\ \wedge\ \textit{condition}_{r-1}(x_1,\cdots,x_{r-1}) \\
\Leftrightarrow & \{\ \text{definition of } \textit{active}_r \text{ for } r>1 \text{ and } x_r = lb_r\ \} \\
& \textit{active}_r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)
\end{array}
$$

  - Induction step ($(x_1,\cdots,x_r-1) \rightarrow (x_1,\cdots,x_r)$, where $x_r > lb_r$): analogously to $x_1-1 \rightarrow x_1$.

Next, we prove that predicate *sig* (Definition 25) is also implemented correctly. We proceed in several steps:

1. The following definition, Definition 34, gives the predicate that corresponds to *sig* in the target program a name: *prg-sig*.
2. Two auxiliary lemmata lead up to Corollary 37, which expresses *prg-sig* analogously to the definition of *sig*.
3. This correspondence is helpful for the proof of Lemma 38, which states that *prg-sig* implies *sig*.

According to Figure 7.10, Part K, a signal is sent if $(k_r(x) \neq 0) \wedge \textit{sigval}_r(x)$. The direction of this signal is $k_r(x)$. This leads to the following definition:

Definition 34 ($\textit{prg-sig}_k^r$). In the target program, a signal is sent into direction k iff the direction given by $k_r(x)$ equals k and $\textit{sigval}_r(x)$ holds. Formally:

$$
\begin{array}{l}
(\forall\ r\ :\ 1 \leq r < d\ :\ (\forall\ (x_1,\cdots,x_r)\ :\ \mathcal{T}(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d) \in \mathcal{TS}\ :\ (\forall\ k\ :\ 0 \leq k < r\ : \\
\qquad \textit{prg-sig}_k^r(x_1,\cdots,x_r) = (k_r(x_1,\cdots,x_r) = k) \wedge \textit{sigval}_r(x_1,\cdots,x_r))))
\end{array}
$$

where $k_r(x)$ is defined in parts E, F and H the target program.

Lemma 35. *If a point $\mathcal{T}(x_1,\cdots,x_d)$ is scanned by the target program, the value of $\textit{sigval}_r$ (second conjunct of Definition 34) is analogous to the first two conjuncts of the definition of sig (Definition 25). Formally:*

$$
\begin{array}{l}
(\forall\ (x_1,\cdots,x_d)\ :\ \mathcal{T}(x_1,\cdots,x_d) \in \mathcal{TS}\ :\ (\forall\ r\ :\ 0 < r \leq d\ :\ \textit{sigval}_r(x_1,\cdots,x_r)\ = \\
\qquad \neg \textit{executed}_r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)\ \wedge \\
\qquad (\forall\ s\ :\ 1 \leq s < r\ :\ \neg \textit{prg-w}_s^r(x_1,\cdots,x_r) \vee \textit{prg-sig}_s^r(x_1,\cdots,x_s-1,\cdots,x_r))\ ))
\end{array}
$$

Proof. Following the program, we distinguish two cases.

- Case 1 ($\textit{prg-active}_r(x_1,\cdots,x_r) \wedge \textit{condition}_r(x_1,\cdots,x_r)$):
  In this case, $\textit{executed}_r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)$ holds:

$$
\begin{array}{ll}
& \mathit{executed}_r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d) \\
\Leftrightarrow & \{ \text{ Definition 16 } \} \\
& \mathit{active}_r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d) \wedge \mathit{condition}_r(x_1,\cdots,x_r) \\
\Leftrightarrow & \{ \text{ Lemma 33 } \} \\
& \mathit{prg\text{-}active}_r(x_1,\cdots,x_r) \wedge \mathit{condition}_r(x_1,\cdots,x_r) \\
\Leftrightarrow & \{ \text{ condition of the first case } \} \\
& \mathit{tt}
\end{array}
$$

Thus,

$$
\begin{array}{ll}
& \neg\mathit{executed}_r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d) \wedge \\
& \quad (\forall\ s : 1 \le s < r : \neg\mathit{prg\text{-}w}^r_s(x_1,\cdots,x_r) \vee \mathit{prg\text{-}sig}^r_s(x_1,\cdots,x_s-1,\cdots,x_r)) \\
\Leftrightarrow & \{ \ \mathit{executed}_r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d),\ \mathit{ff} \text{ is null of } \wedge \ \} \\
& \mathit{ff} \\
\Leftrightarrow & \{ \text{ definition of } \mathit{sigval} \text{ by the program in this case } \} \\
& \mathit{sigval}_r
\end{array}
$$

- Case 2 ($\neg(\mathit{prg\text{-}active}_r(x_1,\cdots,x_r) \wedge \mathit{condition}_r(x_1,\cdots,x_r))$):
  Analogously, in this case, we obtain $\neg\mathit{executed}_r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)$.Then:

$$
\begin{array}{ll}
& \mathit{sigval}_r(x_1,\cdots,x_r) \\
\Leftrightarrow & \{ \text{ definition of } \mathit{sigval} \text{ in this case and definitions in Parts } C \text{ and } G \text{ of} \\
& \quad \text{the program } \} \\
& \neg((\mathit{prg\text{-}w}^r_1(x_1,\cdots,x_r) \wedge \neg\mathit{rcved}^r_1(x_1,\cdots,x_r)) \vee \cdots \vee \\
& \quad \neg(\mathit{prg\text{-}w}^r_{r-1}(x_1,\cdots,x_r) \wedge \neg\mathit{rcved}^r_{r-1}(x_1,\cdots,x_r))) \\
\Leftrightarrow & \{ \text{ de Morgan, twice } \} \\
& (\neg\mathit{prg\text{-}w}^r_1(x_1,\cdots,x_r) \vee \mathit{rcved}^r_1(x_1,\cdots,x_r)) \wedge \cdots \wedge \\
& \quad (\neg\mathit{prg\text{-}w}^r_{r-1}(x_1,\cdots,x_r) \vee \mathit{rcved}^r_{r-1}(x_1,\cdots,x_r)) \\
\Leftrightarrow & \{ \text{ formalization } \} \\
& (\forall\ s : 1 \le s < r : \neg\mathit{prg\text{-}w}^r_s(x_1,\cdots,x_r) \vee \mathit{rcved}^r_s(x_1,\cdots,x_r)) \\
\Leftrightarrow & \{ \text{ definition of } \mathit{rcved} \ \} \\
& (\forall\ s : 1 \le s < r : \neg\mathit{prg\text{-}w}^r_s(x_1,\cdots,x_r) \vee \mathit{prg\text{-}sig}^r_s(x_1,\cdots,x_s-1,\cdots,x_r)) \\
\Leftrightarrow & \{ \ \neg\mathit{executed}_r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d),\ \mathit{tt} \text{ is unit of } \wedge \ \} \\
& \neg\mathit{executed}_r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d) \wedge \\
& \quad (\forall\ s : 1 \le s < r : \neg\mathit{prg\text{-}w}^r_s(x_1,\cdots,x_r) \vee \mathit{prg\text{-}sig}^r_s(x_1,\cdots,x_s-1,\cdots,x_r))
\end{array}
$$

Lemma 36. *If point $\mathcal{T}(x_1,\cdots,x_d)$ is scanned by the target program, the validity of $k_r(x_1,\cdots,x_r) = k$ (first conjunct of Definition 34) implies $\mathit{prg\text{-}sig}^{r-1}_k(x_1,\cdots,x_{r-1})$, in analogy to the third conjunct of the definition of sig (Definition 25). Formally:*

$$
\begin{array}{l}
(\forall\ r : 1 \le r \le d : (\forall\ (x_1,\cdots,x_d) : \mathcal{T}(x_1,\cdots,x_d) \in \mathcal{TS} : (\forall\ k : 0 \le k < r : \\
\quad k_r(x_1,\cdots,x_r) = k \Rightarrow (r = k+1 \vee \mathit{prg\text{-}sig}^{r-1}_k(x_1,\cdots,x_{r-1})))))
\end{array}
$$

Proof.

$$
\begin{array}{ll}
& k_r(x_1,\cdots,x_r) = k \\
\Leftrightarrow & \{ \text{ definition of } k_r \text{ in the program } \} \\
& k_r(x_1,\cdots,x_{r-1},lb_r) = k \\
\Rightarrow & \{ \text{ definition of } k_r \text{ in the program: in Figure 7.8, in Part } F \text{ and in the} \\
& \quad \texttt{else} \text{ branch of part } H\ k_r \text{ is set to } r-1 \text{; in the } \texttt{then} \text{ branch of Part} \\
& \quad H \text{ (where } \mathit{sigval} \text{ holds) it is set to } k_{r-1}(x_1,\cdots,x_{r-1}) \ \ \}
\end{array}
$$

$$
\begin{array}{ll}
& (r-1=k) \vee (sigval_{r-1}(x_1,\cdots,x_{r-1}) \wedge k_{r-1}(x_1,\cdots,x_{r-1})=k) \\
\Leftrightarrow & \{ \text{ Definition 34 } \} \\
& (r=k+1) \vee \mathit{prg\text{-}sig}_k^{r-1}(x_1,\cdots,x_{r-1})
\end{array}
$$

Corollary 37. *If a point $\mathcal{T}(x_1,\cdots,x_d)$ is scanned by the target program, the conjunction of $sigval_r$ and $k_r(x_1,\cdots,x_r)=k$ (and, by Definition 34, the value of $\mathit{prg\text{-}sig}_k^r(x_1,\cdots,x_r)$) is analogous to all three conjuncts of the definition of sig (Definition 25). Formally:*

$$
\begin{array}{c}
(\forall\, r \,:\, 1 \le r \le d \,:\, (\forall\, (x_1,\cdots,x_d) \,:\, \mathcal{T}(x_1,\cdots,x_d) \in \mathcal{TS} \,:\, (\forall\, k \,:\, 0 \le k \le d \,: \\
\mathit{prg\text{-}sig}_k^r(x_1,\cdots,x_r) = \neg executed_r(x_1,\cdots,x_d) \wedge \\
(\forall\, s \,:\, 1 \le s < r \,:\, \neg \mathit{prg\text{-}w}_s^r(x_1,\cdots,x_r) \vee \mathit{prg\text{-}sig}_s^r(x_1,\cdots,x_s-1,\cdots,x_r)) \wedge \\
(r=k+1 \vee \mathit{prg\text{-}sig}_k^{r-1}(x_1,\cdots,x_{r-1}))\ )))
\end{array}
$$

Proof.

$$
\begin{array}{ll}
& \mathit{prg\text{-}sig}_k^r(x_1,\cdots,x_r) \\
\Leftrightarrow & \{ \text{ Definition 34 } \} \\
& sigval_r(x_1,\cdots,x_r) \wedge k_r(x_1,\cdots,x_r)=k \\
\Rightarrow & \{ \text{ Lemma 35 and Lemma 36 } \} \\
& (\neg executed_r(x_1,\cdots,x_d)\ \wedge (\forall\, s \,:\, 1 \le s < r \,:\, \neg \mathit{prg\text{-}w}_s^r(x_1,\cdots,x_r) \vee \\
& \quad \mathit{prg\text{-}sig}_s^r(x_1,\cdots,x_s-1,\cdots,x_r))) \wedge \\
& \quad (r=k+1\ \vee \mathit{prg\text{-}sig}_k^{r-1}(x_1,\cdots,x_{r-1}))
\end{array}
$$

Lemma 38 (*prg-sig* **implements** *sig*)**.**

$$
\begin{array}{c}
(\forall\, r \,:\, 1 \le r \le d \,:\, (\forall\, (x_1,\cdots,x_d) \,:\, \mathcal{T}(x_1,\cdots,x_d) \in \mathcal{TS} \,:\, (\forall\, k \,:\, 0 \le k < r \,: \\
\mathit{prg\text{-}sig}_k^r(x_1,\cdots,x_r) \Rightarrow sig_k^r(x_1,\cdots,x_d))))
\end{array}
$$

Proof. Induction on the "distance" $N = (\Sigma\, i \,:\, 1 \le i \le d \,:\, x_i)$ of point x from the origin

- Induction base ($N=0$):

$$
\begin{array}{ll}
& \mathit{prg\text{-}sig}_k^r(x_1,\cdots,x_r) \\
\Rightarrow & \{ \text{ Definition 34 } (\mathit{prg\text{-}sig}) \ \} \\
& (k_r(x_1,\cdots,x_r)=k) \wedge sigval_r(x_1,\cdots,x_r) \\
\Rightarrow & \{ \text{ predicate calculus } \} \\
& sigval_r(x_1,\cdots,x_r) \\
\Rightarrow & \{ \text{ Lemma 35 } \} \\
& \neg executed_r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d) \wedge \\
& \quad (\forall\, s \,:\, 1 \le s < r \,:\, \neg \mathit{prg\text{-}w}_s^r(x_1,\cdots,x_r) \vee \mathit{prg\text{-}sig}_s^r(x_1,\cdots,x_s-1,\cdots,x_r)) \\
\Rightarrow & \{\ N=0 \text{ implies } (x_1,\cdots,x_r) = (0,\cdots,0) \text{ and } \mathit{prg\text{-}sig}_s^r(x_1,\cdots,x_s- \\
& \quad 1,\cdots,x_r) = \mathit{ff} = sig_s^r(x_1,\cdots,x_s-1,\cdots,x_r,lb_{r+1},\cdots,lb_d)\ \} \\
& \neg executed_r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d) \wedge \\
& \quad (\forall\, s \,:\, 1 \le s < r \,:\, \neg \mathit{prg\text{-}w}_s^r(x_1,\cdots,x_r) \vee sig_s^r(x_1,\cdots,x_s-1,\cdots,x_r,lb_{r+1},\cdots,lb_d)) \\
\Leftrightarrow & \{ \text{ Definition 25 } (sig) \ \} \\
& sig_k^r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)
\end{array}
$$

- Induction step ($N-1 \to N$, where $N>0$):

$prg\text{-}sig^r_k(x_1,\cdots,x_r)$
$\Rightarrow$ { Definition 34 ($prg\text{-}sig$), predicate calculus and Lemma 35, as for the induction base }
$\neg executed_r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)\wedge$
$(\forall\, s \,:\, 1\leq s<r \,:\, \neg prg\text{-}w^r_s(x_1,\cdots,x_r) \vee prg\text{-}sig^r_s(x_1,\cdots,x_s-1,\cdots,x_r))$
$\Rightarrow$ { induction hypothesis for $(x_1,\cdots,x_s-1,\cdots,x_r)$ }
$\neg executed_r(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)\wedge$
$(\forall\, s \,:\, 1\leq s<r \,:\, \neg prg\text{-}w^r_s(x_1,\cdots,x_r) \vee sig^r_s(x_1,\cdots,x_s-1,\cdots,x_r,lb_{r+1},\cdots,lb_d))$
$\Leftrightarrow$ { Definition 25 (sig) }
$sig^r_k(x_1,\cdots,x_r,lb_{r+1},\cdots,lb_d)$

Armed with these lemmata, we can show that the target program executes the body at all points with an inverse image in $\mathcal{X}$.

Remember that the schedule for a nest of while loops is an affine function in the loop indices with positive coefficients. Also, our correctness proof is restricted to such schedules. If we have a mixed nest of for and while loops and the schedule contains a negative coefficient for some for loop indices, we must first re-index the for loop and the corresponding indices in the body to revert the enumeration of the for loop. Then, our methods can be applied.

Lemma 39. *All transformed points with inverse image in $\mathcal{X}$ are scanned if the schedule is an affine function with positive coefficients, i.e., for any dimension t in time,*

$$(\forall\, i \,:\, 1\leq i\leq d \,:\, \mathcal{T}_{t,i}\geq 0) \Rightarrow (\forall\, x \,:\, x\in\mathcal{X} \,:\, \mathcal{T}x\in\mathcal{TS}).$$

Proof. Our aim is to apply Definition 30. First, we find a point $\widehat{x}\in\widehat{\mathcal{X}}$ with $t(x)<_{\text{lex}} t(\widehat{x})$:

$x\in\mathcal{X}$
$\Rightarrow$ { Definition 17 and Definition 16 }
$active_d(x) \wedge condition_d(x)$
$\Leftrightarrow$ { Definition 15 for $(x_1,\cdots,x_r+1,\cdots,x_d)$, where $1\leq r\leq d$ is a while dimension }
$active_r(x_1,\cdots,x_r+1,\cdots,x_d)$
$\Rightarrow$ { Definition 18 }
$(x_1,\cdots,x_r+1,\cdots,x_d)\in\widehat{\mathcal{X}}$

We name this point $\widehat{x}$; $0 <_{\text{lex}} t(\widehat{x})-t(x)$, since r is a while loop.

Now we can prove the lemma. Let x be any point in $\mathcal{X}$ and $\widehat{x}$ a corresponding point in $\widehat{\mathcal{X}}$ as just defined. Applying Definition 30, we show that all points that are enumerated by time slices up to and including $t(p)$ do not abort the program:

tt
$\Leftrightarrow$ { trivial quantification }
$(\forall\, x' \,:\, x'\in\mathcal{I}\wedge t(x')\leq_{\text{lex}} t(x) \,:\, t(x')\leq_{\text{lex}} t(x))$
$\Rightarrow$ { property $t(x)<_{\text{lex}} t(\widehat{x})$ of $\widehat{x}$ and transitivity of $\leq_{\text{lex}}$ and $<_{\text{lex}}$ }
$(\forall\, x' \,:\, x'\in\mathcal{I}\wedge t(x')\leq_{\text{lex}} t(x) \,:\, t(x')<_{\text{lex}} t(\widehat{x}))$
$\Leftrightarrow$ { t is an affine function, say, row t in $\mathcal{T}$ }
$(\forall x' \,:\, x'\in\mathcal{I}\wedge t(x')\leq_{\text{lex}} t(x) \,:$
$(\exists\, t \,:\, 1\leq t\leq d \,:\, (\Sigma\, i \,:\, 1\leq i\leq d \,:\, \mathcal{T}_{t,i}x'_i) < (\Sigma\, i \,:\, 1\leq i\leq d \,:\, \mathcal{T}_{t,i}\widehat{x}_i)))$
$\Rightarrow$ { $(\forall\, t,i \,:\, 1\leq t,i\leq d \,:\, \mathcal{T}_{t,i}\geq 0)$ by assumption, arithmetic }

$$
\begin{array}{ll}
& (\forall\, x' \;:\; x' \in \mathcal{I} \wedge t(x') \leq_{\text{lex}} t(x) \;:\; (\exists\, i \;:\; 1 \leq i \leq d \;:\; x'_i < \widehat{x}_i)) \\
\Leftrightarrow & \quad \{ \text{ negation of Definition 24 } \} \\
& (\forall\, x' \;:\; x' \in \mathcal{I} \wedge t(x') \leq_{\text{lex}} t(x) \;:\; \neg m_0^d(x')) \\
\Rightarrow & \quad \{ \text{ contrapositive of Lemma 27 } \} \\
& (\forall\, x' \;:\; x' \in \mathcal{I} \wedge t(x') \leq_{\text{lex}} t(x) \;:\; \neg sig_0^d(x')) \\
\Rightarrow & \quad \{ \text{ Lemma 38 and Definition 34 } \} \\
& (\forall\, x' \;:\; x' \in \mathcal{I} \wedge t(x') \leq_{\text{lex}} t(x) \;:\; \neg(k_d(x') = 0) \wedge sigval_d(x')) \\
\Rightarrow & \quad \{ \text{ Definition 30 } \} \\
& \textit{prg-scanned}(x) \\
\Leftrightarrow & \quad \{ \text{ Definition 30 } \} \\
& \mathcal{T}x \in \mathcal{T}\mathcal{S}
\end{array}
$$

Lemma 40. *prg-active ensures that all legal points are executed, i.e., that $\mathcal{T}x \in \mathcal{T}\mathcal{I}$ is executed iff $x \in \mathcal{I}$ is executed and $\mathcal{T}x$ is scanned by the target program. Formally:*

$$(\forall\, x \;:\; x \in \mathcal{I} \;:\; \textit{prg-exec}(x) \Leftrightarrow \textit{prg-scanned}(x) \wedge \textit{executed}(x)).$$

Proof.

$$
\begin{array}{ll}
& \textit{prg-exec}(x) \\
\Leftrightarrow & \quad \{ \text{ Definition 31 } (\textit{prg-exec}) \text{ } \} \\
& \textit{prg-scanned}(x) \wedge \textit{prg-active}_d(x_1, \cdots, x_d) \wedge \textit{condition}_d(x_1, \cdots, x_d) \\
\Leftrightarrow & \quad \{ \text{ Lemma 33 } \} \\
& \textit{prg-scanned}(x) \wedge \textit{active}_d(x_1, \cdots, x_d) \wedge \textit{condition}_d(x_1, \cdots, x_d) \\
\Leftrightarrow & \quad \{ \text{ Definition 16 } (\textit{executed}) \text{ } \} \\
& \textit{prg-scanned}(x) \wedge \textit{executed}(x_1, \cdots, x_d)
\end{array}
$$

Theorem 41. *For affine schedules with positive coefficients, the loop body b is executed at $y \in \mathcal{T}\mathcal{I}$ iff it is executed at $\mathcal{T}^{-1}y \in \mathcal{I}$. Formally, for any dimension t in time:*

$$(\forall\, i \;:\; 1 \leq i \leq d \;:\; \mathcal{T}_{t,i} \geq 0) \Rightarrow (\forall\, y \;:\; y \in \mathcal{T}\mathcal{I} \;:\; \textit{prg-exec}(\mathcal{T}^{-1}y) \Leftrightarrow \textit{executed}(\mathcal{T}^{-1}y))$$

Proof.

"$\Rightarrow$": part "$\Rightarrow$" of Lemma 40.

"$\Leftarrow$":

- y is scanned (Lemma 39);
- every scanned point whose inverse image is in $\mathcal{X}$ is executed (part "$\Leftarrow$" of Lemma 40).

7.3.4.3 Possible Adaptations of the Code to the Target Architecture

Memory reduction for distributed memory systems. For a real implementation, we first introduce re-assignments by a simple modification of the skeleton in Figures 7.8 to 7.10 (we assume an injective allocation):

- all variables that are indexed with $(x_1, \cdots, x_r)$ become local scalars, e.g., $\textit{prg-active}_r(x_1, \cdots, x_r)$ becomes $\textit{prg-active}_r$;

- all variables that are indexed with $(x_1, \cdots, x_r{+}1)$ become local renamed scalars, e.g., $\textit{prg-active}_r(x_1, \cdots, x_r{+}1)$ becomes $\textit{prg-active\_out}_r$;
- all variables that are indexed with $(x_1, \cdots, x_r, lb_{r+1})$ become local scalars, e.g., $\textit{prg-active}_{r+1}(x_1, \cdots, x_r, lb_{r+1})$ becomes $\textit{prg-active}_{r+1}$ (note the different index).

Note, that the upper bound d on r is known at compile time.

Adaptation for asynchronous systems. As we mentioned already, the signaling scheme is most easily described for synchronous machines. In this case, the given target code is complete and correct.

In the asynchronous case, we can always find a space-time mapping that is scannable (Section 5.2.6). Still, for whatever reasons, one has the option of an unscannable transformation also for the asynchronous model, but with a slightly modified version of the target code just presented.

The modifications result from the fact that, in the asynchronous case, there is no global clock, i.e., the time component of every space-time mapped iteration cannot be interpreted globally. Thus, sending any message from an iteration on processor P at time t to another iteration on processor P' with execution time t' makes no sense—t' might be in the past with respect to the clock of processor P.

We can avoid this problem as follows: instead of the conditional sending of valueless signals, send unconditionally messages carrying the value of the condition, and use the blocking receive for the receipt of these messages. The modified part K is

```
if k_r(x_1, ..., x_r) ≠ 0 then
    asend(chan^S_{k_r}, [sigval_r(x_1, ..., x_r)])
endif
```

Note, however, the increase in the number of messages.

Adaptation for shared memory systems. Section 7.2 contains a termination detection scheme for shared memory systems via counters for shared memory. Alternatively, the signaling scheme for distributed memory just derived can also be adapted to shared memory systems: of course, parts B and J are superfluous. However, the exchange of the signals in parts A and K must be transformed to shared memory access.

Let us briefly compare the signaling scheme adapted to shared memory and the counter scheme with the extreme optimization of using one counter per tooth (i.e., $r = 1$ in Section 7.2.4). The optimized counter scheme still has to deal with conflicting accesses to the counters which leads to an increase in execution time; this kind of bottleneck does not exist in the adapted signaling scheme (at least not, if we neglect possible conflicts on the memory bus of the shared memory system). On the other hand, the signaling scheme may increase the execution time due to the necessary prolongation of the teeth. For both schemes, the order of magnitude of the increase of the execution time is the same: linear in the extent of the loops.

7.3.5 The Example

The manual application of the signaling scheme to our example program is error-prone and, as of yet, we have no implementation of it. Thus, we do not present the target code here, but offer only some remarks for the implementation of the signaling scheme.

We have presented our scheme for perfect loop nests. For an imperfect nest there are two options: either one uses a separate instance of the signaling scheme for every statement or one uses a single instance of the scheme for the whole loop nest. In the first option, our scheme can be applied without change; however, it would result in more communications than necessary. Therefore, we propose to implement the second option.

If we want to use only one instance of the signaling scheme for the whole program, we must distinguish between loop statements and regular statements: the signaling scheme is implemented for the loop statements only, i.e., every loop statement at any level r computes the values of its predicate $active_r$, which are passed on to all statements (regular and loop statements) in the body of the loop. Thus, the main modification is that the recursive definition of $active_d$ is not unrolled at a single iteration point (as in Figure 7.8), but is evaluated recursively, with different iteration points computing the parts of $active_d$ at the different recursion levels r caused by alternative (2) of Definition 15.

Chapter 8

LooPo

This chapter presents an overview and the current state of our source-to-source parallelizer LooPo (**Loo**p Parallelization in the **Po**lyhedron Model). Since LooPo is not yet complete, we cannot offer the reader any performance measures of the developed target programs on real parallel machines at this time.

LooPo is a prototype system whose purpose is to assist us in the research on and evaluation of space-time mapping methods for loop parallelization. To that end, it implements the complete path from executable source code to executable target code, with switches for choosing alternative methods. At present, we provide several inequation solving methods, several dependence analyzers, schedulers, allocators and several methods of code generation.

LooPo is in the public domain and uses only freely available software to ensure easy distribution. It runs on Sun workstations under SunOS 4.1.x and Solaris 2.x, and on PCs under Linux.

LooPo can be used as a platform for experimenting with any step of the parallelization process in the model; anybody interested in one special aspect of the parallelization can plug his own module to LooPo and gets a complete source-to-source compiler. The central data structures of the interface (restricting the applicability) are—according to the method—polyhedra and piecewise affine functions.

However, the most important aspect of LooPo is that it integrates while loops. More details on this aspect are given in Section 8.3.

8.1 The Structure of LooPo

LooPo traverses a sequence of steps which transform the source program to an executable parallel target program. There are modules for scanning and parsing, (in)equality solving, dependence analysis, scheduling, allocation and target code generation. A front end provides the user with a graphical interface by which he/she can control LooPo. There is also a graphical tool for displaying index spaces and iteration dependence graphs of loop nests.

Subsequently, we give a very brief overview of the system since it is not at the center of this thesis. See our Web pages on LooPo for more details [41]. Also a list of all people working in the LooPo team can be found there—all implementation has been done via programming projects and master's theses of students at the University of Passau.

8.1.1 The Front End

The front end enables the user to invoke modules by mouse clicks (Figure 8.1). In order to optimize LooPo's results and suit the needs of the user, most modules provide an option window where specific features can be altered (Figure 8.2).

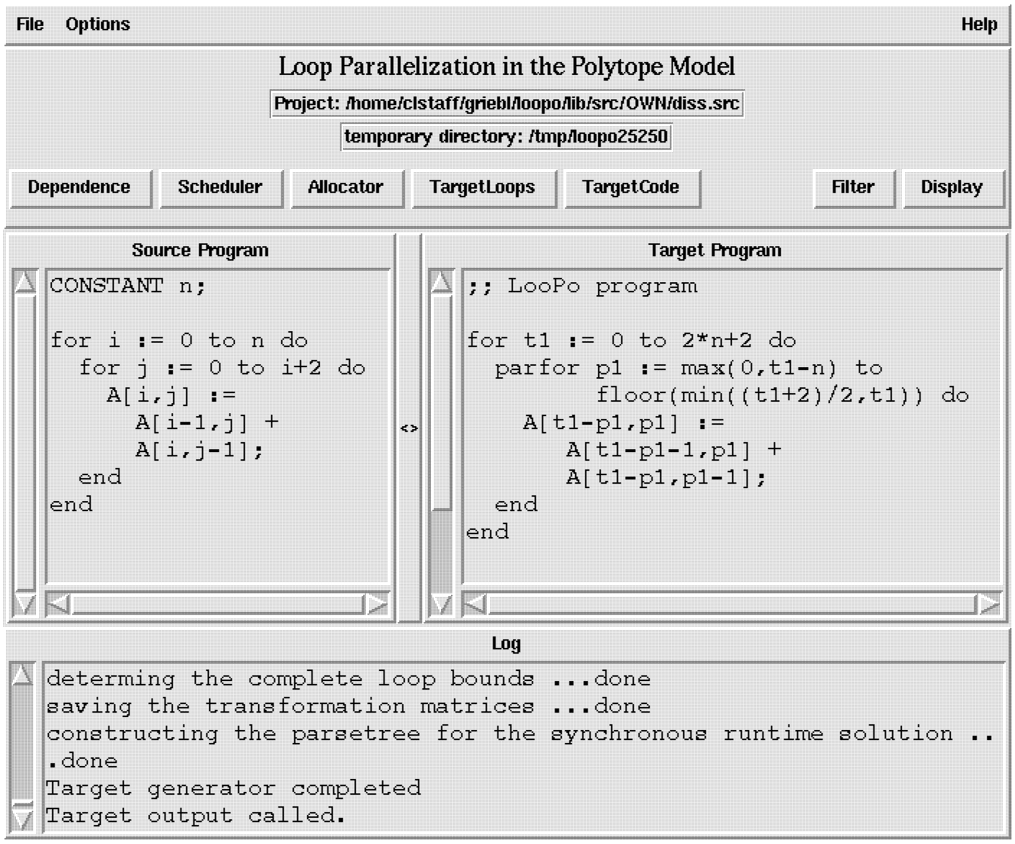

Figure 8.1: LooPo's main window

8.1.2 The Input to LooPo

LooPo accepts (imperfect) loop nests in C and Fortran notation (among others) and declarations of functions, procedures and symbolic constants. LooPo also takes explicit specifications of dependences, schedules and allocations by the user, if so desired. By stating explicit dependences, one can experiment with the space-time mapping of non-executable programs, i.e., programs with incomplete loop bodies.

8.1.3 The Inequation Solvers

There are several methods for parametric linear programming, which is the central mathematical problem of the polyhedron model. We considered the following methods for use in LooPo:

1. *Fourier-Motzkin.* This is the standard doubly exponential method of polytope projection (Section 3.3.1).

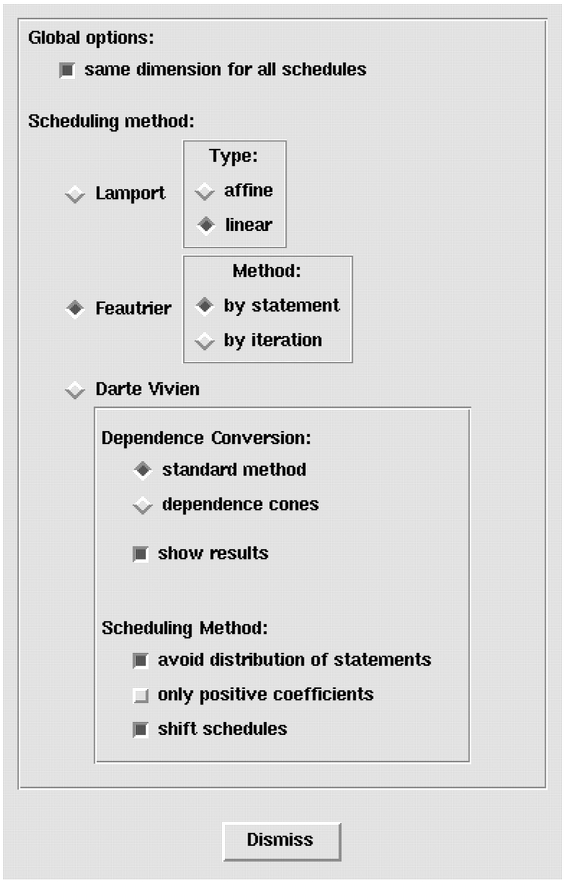

Figure 8.2: LooPo' options window for the scheduler

2. *PIP.* This is Feautrier's system for parametric integer programming [27]. It proceeds indirectly by transforming the original system of inequations into a dual system and solving that. In principle it is an extension of the well-known simplex algorithm so as to deal with parameters.

3. *Weispfenning.* There is another direct method which is only singly exponential [60]. It performs better than Fourier-Motzkin on problems with more than four variables.

4. *Omega.* The Omega library [48] by Pugh solves linear programs on the basis of Presburger formulas (affine constraints, the usual logical connectives, and existential and universal quantifiers), with efficient heuristics for this special application field.

The current implementation of LooPo uses mainly PIP; the dependence module offers a choice of PIP or Fourier-Motzkin. Omega will be integrated as an alternative for PIP and Fourier-Motzkin in all modules.

8.1.4 The Dependence Analyzers

At present, LooPo features two tools for dependence analysis:

1. *Banerjee.* The dependence analysis method described by Banerjee [4] makes no distinction between flow, anti and output dependences. Furthermore, spurious dependences are not eliminated.

2. *Feautrier.* The method of Feautrier [28] allows potentially more parallelism, since it only considers true dependences (no anti and output dependences)—thus, enforcing a conversion to single-assignment form—and eliminates all spurious dependences, i.e., it computes only the flow dependences—however for affine loops only.

8.1.5 The Schedulers

Presently, LooPo provides three different automatic schedulers:

1. *Lamport.* The hyperplane method by Lamport [5, 39] can handle perfectly nested for loops with uniform dependences. It yields a one-dimensional affine schedule for the complete loop body and, as allocation, a projection onto the source axes such that the space-time mapping formed by the combination of schedule and allocation is unimodular.

2. *Feautrier.* The Feautrier scheduler [29, 30] determines an optimal (concave) schedule for imperfectly nested for loops with affine dependences, at the cost of a longer computation time based on the necessity of dealing with parametric integer linear programming [27]. The resulting schedule for every statement can be multi-dimensional and piecewise affine.

   For a comparison with Lamport's method, one can call the Feautrier scheduler *by iteration* (in the case of a perfectly nested input program), which enforces the same schedule for all statements in the loop body.

3. *Darte/Vivien.* Darte and Vivien proposed a fast scheduler with reasonably good results [20], which can schedule arbitrary loop programs with uniform and non-uniform dependences. It uses a less precise dependence description (direction vectors) than the Feautrier scheduler. Therefore, the quality of its schedules is somewhere between that of Lamport's and Feautrier's schedules.

8.1.6 The Allocators

Presently, LooPo provides two different allocators:

1. *Feautrier.* Feautrier's method [31] determines the placement of operations on virtual processors. It is based on the "owner computes rule" and tries to "cut" dependences by mapping the depending operations to the same processor, starting with dependences in the highest dimensions (greedy heuristic).

2. *Dion/Robert.* The method of Dion and Robert [25] uses the reduced dependence graph, where the dependences are either given by the direction vectors or the dependence cone. In addition to the allocation for the computation an allocation of the data, i.e., a data distribution is generated.

The allocators do not inspect the schedule, and may therefore generate an allocation in which some dimensions are linearly dependent on the schedule.

In addition, we are currently adding a module for partitioning to LooPo. This module maps the virtual processors to a fixed number of real processors.

8.1.7 The Display Module

LooPo also features a graphical displayer which depicts the source index space and the dependences therein in up to three dimensions. In the current implementation all statements must have the same index space; this excludes imperfectly nested loops. A dependence filter provides a graphical interface to enable the user to select a subset of statements and dependences which satisfy these restrictions.

The displayed polytope can be rotated or even transformed by an arbitrary affine matrix to show the target space.

8.1.8 The Target Generator

The target generator consists of two modules: one derives the target loop nest(s), the other adds communications for synchronization and communication.

8.1.8.1 The Target Loops

The loops of the transformed source program are constructed from the index spaces, the dependences, the schedule and the allocation. Note that transformations can be individual for every statement in the source program. The target loops are represented as a parse tree which does not contain any synchronization or communication statements.

The construction of the parse tree proceeds in two phases. First the program parts are constructed and transformed individually, and then the results are combined to a single target program, as described in Section 3.3.2. Aside from the two options of synchronous or asynchronous code, three merging strategies are available [61]:

1. the parts are simply combined with a *parallel operator*, i.e., there are several separate loop nests which are assumed to be executed in parallel,
2. merging at run time as described in Section 3.3.2,
3. merging at compile time.

8.1.8.2 Synchronization and Communication

The parse tree representing the target loops is then translated to one of a variety of possible output languages, e.g., some parallel C or Fortran dialect or PARIX-C. Synchronization and communication is added if the user so desires [26]. The target program (with communication) is executable on any PARIX machine.

8.2 First Experiences

Our first tests showed that there are two main restrictions limiting the applicability of LooPo in practice. The first is the lack of conditional statements in the current version of LooPo. This will be fixed soon.

The other limitation is more deeply connected with the use of the polytope model for space-time mapping. The polytope model offers very precise analysis and scheduling techniques. However, these techniques are based on integer linear programming which is a computationally complex problem. We have had to learn that the (in)equation solvers are the most problematic component of the parallelizer: not only do they consume most of the compilation time but they even frequently fail to compute a solution in real applications.

8.3 LooPo and while Loops

One of the main reasons for starting the project LooPo was our need for a parallelizing source-to-source compiler whose internal structure we know very well, in order to be able to implement parallelization techniques for while loops. The extension to loops of Class 2, 1 and 0 will be part of version 2 of LooPo which we hope to complete by the end of 1996.

Chapter 9

Conclusions

The contribution of this thesis is an extension of the applicability of parallelization methods. We started with the polytope model, which is a very useful mathematical framework for automatic parallelization, but which is restricted to for loops with affine bounds. We succeeded in generalizing the methods for the polytope model and developed, in several stages, the polyhedron model as a mathematical framework for the parallelization of loop nests containing while loops.

First, we decided to use an index for while loops, in analogy to for loops, and dropped the requirement that index spaces must be bounded, which did not affect the space-time mapping techniques. Then, we realized that the execution spaces at run time are, in general, not convex, leading to target execution spaces which cannot be scanned precisely. We distinguished space-time mappings that do not raise this problem and suggested, for the other space-time mappings, a scheme which prevents the execution of holes in the target execution space. Finally we bounded the dimensions in space by partitioning and the dimensions in time, depending on the target machine, by various termination detection schemes.

With all these schemes at our disposal, we can drop the requirement of affinity on loops altogether.

However, there is, of course, an important difference in efficiency: first, while loops always lead to a loop-carried dependence, thus reducing parallelism. Second, and probably worse, these dependences come from the necessity of transferring information between different points of the index space, which leads to many communications. Third, the treatment of unscannable spaces, necessary for arbitrary for loops as well as for while loops, results in a constant slowdown due to the necessity of evaluating guards at every scanned target index point.

On the other hand, we have seen that the parallelism in nested while loops may offer the potential for a speed-up of orders of magnitude: if there are only while dependences, one dimension in time is sufficient, i.e., we can reach linear time. Of course, additional dependences in the loop body may reduce the parallelism further.

Note that maximal parallelism does not imply maximal efficiency of the parallel program; this observation, also valid for for loops, is still more important for while loops because of their increased communication volume. Therefore, partitioning is an important subject in parallelizing nested while loops.

Besides extending the applicability of existing parallelization methods, we also have suggested a classification of loops. Table 9.1 gives an idea of the impact of each class on code

Class	Transformation				Comments
	scannable		unscannable		
	guard	bound	guard	bound	
4	none	arith	none	arith	polytope model
3	none	arith	none	arith	no general mathematical methods
2	none	arith	arith	arith	
1	none	iter	iter	scheme	special cases exist
0	none	iter	iter	scheme	

Table 9.1: The impact of classes of loops and scannability to code generation

generation, for both scannable and unscannable transformations. In each case, the complexity of the code generation is determined by the nature of the guards, if any, and the form of the loop bounds.

For the guards we distinguish:

- none local guards are not necessary,
- arith the guard is an arithmetic expression,
- iter the guard must be computed iteratively.

For the bounds we distinguish:

- arith the loop bound is an arithmetic expression, similarly to the source program,
- iter the loop bound must be computed iteratively, similarly to the source program,
- scheme termination detection must be performed by a special scheme.

Note that we have discussed a simple speculative scheme in the case of robust and strict conditions in loops of Class 1, which does not appear in the table. Note further, that guards may be introduced due to merging program parts at run time, due to partitioning or due to the fact that loop statements of while loops become regular statements inside the loop body—even if the entry in the table is "none".

Our work does not deal with speculation in the general case. One reason is that we wanted to avoid very low-level problems for code generation on the technical side, as, e.g., handling arithmetic exceptions in speculatively executed iterations, as well as the exploitation of algorithm-specific properties on the abstract side, such as convergence properties and numerical stability, because we are interested in a machine-independent general-purpose method for parallelizing loop nests containing while loops.

The other reason is that, of course, also in the speculative approach, target loops must be generated. We expect that our methods can at least be a basis for that purpose. One minor difference to the presented code generation schemes is that, for speculative execution, some of the local guards can be dropped; this means that holes are assigned useful work (even if that work is not part of the source program, e.g., additional iterations of an approximation algorithm). The major problem will probably be to find adequate termination conditions for speculative execution.

If such problems do not occur, as, e.g., in robust and strict loops of Class 1, one might drop (or at least replace) some carefully selected control dependences in order to increase parallelism and use our scheme again.

Combining the speculative and the conservative approach in one common framework is interesting future work.

Of course, there still remains a lot of (technical) work to be done. In this thesis, we restricted the technical discussions to perfect nests of only while loops in the intent of a clean presentation. Our main concern was to show that a general loop nest can be parallelized at all, and at which costs; we have reached this goal since one can (e.g., with the help of guards) transform any loop program to a perfect nest of while loops. However, in practice the central goal is the efficiency of the parallel program. Thus, we must not transform a program to a perfect nest of while loops but we must exploit any possibility for optimization offered by each individual loop.

Furthermore, current partitioning techniques are optimized for nests of affine loops. Since these techniques cannot be used for while loops, we can offer only a suboptimal solution at present. The importance of partitioning in the presence of while loops certainly justifies the search for optimal partitioning techniques for loop nests containing while loops.

The central remaining limitation of the polyhedron model is the restriction to arrays as the only data structure, which is inherited from the polytope model. Efforts to relax this restriction are currently being undertaken. Progress in this area would eliminate the necessity of manual interaction during program analysis. This would allow us to run our methods on a wide range of applications completely automatically.

We expect that one major field of application is the parallelization of algorithms for sparse data structures, since sparsity usually leads to irregularity. As seen in our example of computing the reflexive transitive closure of a sparse graph, this kind of algorithm can be parallelized without speculation, i.e., our methods can be applied without change.

Bibliography

[1] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Proc. 3rd ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming (PPoPP)*, pages 39–50. ACM Press, 1991.

[2] F. Balasa, F. Franssen, F. Catthor, and H. De Man. Transformation of nested loops with modulo indexing affine recurrences. *Parallel Processing Letters*, 4(3):271–280, September 1994.

[3] U. Banerjee. *Dependence Analysis for Supercomputing.* The Kluwer Int. Series in Engineering and Computer Science: Parallel Processing and Fifth Generation Computing. Kluwer, 1988.

[4] U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer, 1993.

[5] U. Banerjee. *Loop Transformations for Restructuring Compilers: Loop Parallelization.* Kluwer, 1994.

[6] M. Barnett and C. Lengauer. Unimodularity and the parallelization of loops. *Parallel Processing Letters*, 2(2–3):273–281, 1992.

[7] A. Bernstein. Analysis of programs for parallel processing. *IEEE Trans. on Electronic Computers*, EC-15(5):757–763, October 1966.

[8] J. P. Bonomo and W. R. Dykson. Pipelined iterative methods for shared-memory machines. *Parallel Computing*, 11:187–199, 1989.

[9] Z. Chamski. Scanning polyhedra with DO loop sequences. In B. Sendov and I. Dimov, editors, *Proc. Workshop on Parallel Architectures (WPA 92).* Elsevier (North-Holland), 1992.

[10] P. Clauss, C. Mongenet, and G. R. Perrin. Calculus of space-optimal mappings of systolic algorithms on processor arrays. *J. VLSI Signal Processing*, 4(1):27–36, February 1992.

[11] J.-F. Collard and M. Griebl. Generation of synchronous code for automatic parallelization of while loops. In S. Haridi, K. Ali, and P. Magnusson, editors, *EURO-PAR '95 Parallel Processing*, Lecture Notes in Computer Science 966, pages 315–326. Springer-Verlag, August 1995.

[12] J.-F. Collard. Code generation in automatic parallelizers. In C. Girault, editor, *Proc. Int. Conf. on Applications in Parallel and Distributed Computing, IFIP W.G. 10.3*, pages 185–194. North-Holland, April 1994.

[13] J.-F. Collard. A method for static scheduling of dynamic control programs. Technical Report 94-34, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, December 1994.

[14] J.-F. Collard. Space-time transformation of while-loops using speculative execution. In *Proc. 1994 Scalable High Performance Computing Conf.*, pages 429–436. IEEE Computer Society Press, May 1994.

[15] J.-F. Collard. Automatic parallelization of `while`-loops using speculative execution. *Int. J. Parallel Programming*, 23(2):191–219, 1995.

[16] J.-F. Collard, D. Barthou, and P. Feautrier. Fuzzy array dataflow analysis. In *Proc. 5th ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming (PPoPP)*, pages 92–102. ACM Press, July 1995.

[17] J.-F. Collard and P. Feautrier. Automatic generation of data parallel code. In H. J. Sips, editor, *Proc. Fourth International Workshop on Compilers for Parallel Computers*, pages 321–332, December 1993.

[18] A. Darte. Regular partitioning for synthesizing fixed-size systolic arrays. *INTEGRATION*, 12(3):293–304, December 1991.

[19] A. Darte and Y. Robert. Constructive methods for scheduling uniform loop nests. *IEEE Trans. on Parallel and Distributed Systems*, 5(8):814–822, August 1994.

[20] A. Darte and F. Vivien. Automatic parallelization based on multi-dimensional scheduling. Technical Report 94-24, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, September 1994.

[21] A. Darte and Y. Robert. Mapping uniform loop nests onto distributed memory architectures. *Parallel Computing*, 20(5):679–710, May 1994.

[22] A. Darte and Y. Robert. Affine-by-statement scheduling of uniform and affine loop nests over parametric domains. *J. Parallel and Distributed Computing*, 29(1):43–59, August 1995.

[23] A. Darte and F. Vivien. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. In *Parallel Architectures and Compilation Techniques*. Computer Science Press, 1996.

[24] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[25] M. Dion and Y. Robert. Mapping affine loop nests: New results. In *Lecture Notes in Computer Science 919*, pages 184–189. Springer-Verlag, 1995.

[26] P. Faber. Transformation von Shared-Memory-Programmen in Distributed-Memory-Programme. Master's thesis, Fakultät für Mathematik und Informatik, Universität Passau, 1996. To appear in December.

[27] P. Feautrier. Parametric integer programming. *Operations Research*, 22(3):243–268, 1988.

[28] P. Feautrier. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming*, 20(1):23–53, February 1991.

[29] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part I. One-dimensional time. *Int. J. Parallel Programming*, 21(5):313–348, October 1992.

[30] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *Int. J. Parallel Programming*, 21(6):389–420, October 1992.

[31] P. Feautrier. Toward automatic distribution. *Parallel Processing Letters*, 4(3):233–244, 1994.

[32] A. Fernández, J. Llabería, and M. Valero-García. Loop transformation using nonunimodular matrices. *IEEE Trans. on Parallel and Distributed Systems*, 6(8):832–840, August 1995.

[33] M. Geigl. Parallelization of general loop nests in the polyhedron model. Master's thesis, Fakultät für Mathematik und Informatik, Universität Passau, 1996. To appear in December.

[34] I. Graham and T. King. *The Transputer Handbook*. Prentice-Hall, 1990.

[35] Hyperparallel Technologies, Ecole Polytechnique Projet X-Pôle 91128 Palaiseau Cedex France. *Hyper C Documentation*, June 1993.

[36] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590, July 1967.

[37] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. Technical Report CS-TR-3317, Dept. of Computer Science, Univ. of Maryland, 1994.

[38] S.-Y. Kung. *VLSI Processor Arrays*. Prentice-Hall Int., 1988.

[39] L. Lamport. The parallel execution of DO loops. *Comm. ACM*, 17(2):83–93, February 1974.

[40] C. Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR'93*, Lecture Notes in Computer Science 715, pages 398–416. Springer-Verlag, 1993.

[41] LooPo. http://www.uni-passau.de/~loopo/.

[42] A. Martin. The probe: An addition to communication primitives. *Information Processing Letters*, 20(3):125–130, 1985.

[43] V. Maslov and W. Pugh. Symplifying polynomial constraints over integers to make dependence analysis more precise. In B. Buchberger and J. Volkert, editors, *Parallel Processing: CONPAR 94 – VAPP VI*, Lecture Notes in Computer Science 854, pages 737–748. Springer-Verlag, 1994.

[44] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, 1988.

[45] Parsytec. *PARIX 1.2 Reference Manual*, March 1993.

[46] H. Partsch. Some experiments in transforming towards parallel executability. In R. Paige, J. Reif, and R. Wachter, editors, *Parallel Algorithm Derivation and Program Transformation*, pages 71–110. Kluwer Academic Publishers, 1993.

[47] W. Pugh. A practical algorithm for exact array dependence analysis. *Comm. ACM*, 35(8):102–114, August 1992.

[48] W. Pugh and D. Wonnacott. Eliminating false data dependences using the Omega test. *ACM SIGPLAN Notices*, 27(7):140–151, July 1992. *Proc. ACM SIGPLAN '92 Conference on Programming Language Design and Implementation (PLDI).*

[49] W. Pugh and D. Wonnacott. Going beyond integer programming with the Omega test to eliminate false data dependences. Technical Report CS-TR-3191, Dept. of Computer Science, University of Maryland, College Park, December 1992.

[50] P. Quinton. The systematic design of systolic arrays. In F. F. Soulié, Y. Robert, and M. Tchuente, editors, *Automata Networks in Computer Science*, chapter 9, pages 229–260. Manchester University Press, 1987. Also: Technical Reports 193 and 216, IRISA (INRIA-Rennes), 1983.

[51] P. Quinton and V. van Dongen. The mapping of linear recurrence equations on regular arrays. *J. VLSI Signal Processing*, 1(2):95–113, October 1989.

[52] S. V. Rajopadhye. Synthesizing systolic arrays with control signals from recurrence equations. *Distributed Computing*, 3:88–105, 1989.

[53] S. K. Rao. *Regular Iterative Algorithms and their Implementations on Processor Arrays.* PhD thesis, Department of Electrical Engineering, Stanford University, October 1985.

[54] S. K. Rao and T. Kailath. Regular iterative algorithms and their implementations on processor arrays. *Proc. IEEE*, 76(3):259–282, March 1988.

[55] A. Schrijver. *Theory of Linear and Integer Programming.* Series in Discrete Mathematics. John Wiley & Sons, 1986.

[56] M. Schumergruber. Partitionierung von parallelen Schleifensätzen. Master's thesis, Fakultät für Mathematik und Informatik, Universität Passau, 1997. To appear in February 1997.

[57] J.-P. Sheu and T.-H. Tai. Partitioning and mapping nested loops on multiprocessor systems. *IEEE Trans. on Parallel and Distributed Systems*, 2:430–439, 1991.

[58] J. Teich and L. Thiele. Partitioning of processor arrays: A piecewise regular approach. *INTEGRATION*, 14(3):297–332, 1993.

[59] P. P. Tirumalai, M. Lee, and M. S. Schlansker. Parallelization of while loops on pipelined architectures. *J. Supercomputing*, 5:119–136, 1991.

[60] V. Weispfenning. Parametric linear and quadratic optimization by elimination. Technical Report MIP-9404, Fakultät für Mathematik und Informatik, Universität Passau, 1994. To appear in *J. Symbolic Computation.*

[61] S. Wetzel. Automatic code generation in the polytope model. Master's thesis, Fakultät für Mathematik und Informatik, Universität Passau, 1995.

[62] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* Research Monographs in Parallel and Distributed Computing. MIT Press, 1989.

[63] M. Wolfe. The Tiny loop restructuring research tool. In H. D. Schwetman, editor, *Proc. Int. Conf. on Parallel Processing*, volume II, pages 46–53. CRC Press, 1991.

[64] Y. Wu and T. G. Lewis. Parallelizing while loops. In D. A. Padua, editor, *Proc. Int. Conf. on Parallel Processing*, volume II, pages 1–8. Pennsylvania State University Press, 1990.

[65] J. Xue. Automating non-unimodular transformations of loop nests. *Parallel Computing*, 20(5):711–728, May 1994.

Index