UNIVERSITÄT PASSAU
Fakultät für Mathematik und Informatik

Parallelization of Loop Nests with General Bounds in the Polyhedron Model

Diplomarbeit am
Lehrstuhl für Programmierung
Prof. C. Lengauer, Ph.D.

Autor: Max Geigl
Betreuer: Dr. Martin Griebl

März 1997

Abstract

The polytope model is one possible (mathematical) basis for parallelizing sequential computer programs automatically. It proved to be well suited for the parallelization of loop nests containing only for loops whose bounds satisfy several restrictions. Recent research efforts propose an extension of the polytope model, the polyhedron model, and provide an implementation for perfectly nested while loops.
The first part of this thesis examines the implications of different loop types for target code generation and integrates the results in a class hierarchy. The second part extends the polyhedron model to imperfect loop nests containing general for loops, while loops and if statements. We provide also one possible implementation.

Acknowledgments

Not surprisingly this diploma thesis has not been developed in a secluded chamber, shielded from any (good or bad) influences. Now that the work is done, I would like to say a heart-felt "THANK YOU" to all the people who have had a positive effect on the thesis itself as well as on the "working climate".

In particular, I would like to mention the head of the Lehrstuhl für Programmierung, Professor C. Lengauer. He gave me the opportunity to write this thesis in English, which would not have been possible without his thorough proofreading.

Dr. Martin Griebl was my tutor and is the manager of the LooPo project. Martin, thank you for waking my interest in "multi-dimensional combs", for the discussions, hints, suggestions, encouragements, the "Auto-Card" and your patience — even when I disturbed in the mornings and although you had to prepare for your defense. Thank you also for accompanying me (not only on the way through my study at Passau) like a big brother.

I always felt in good hands with the *LooPo team whose members, at that time, were Sven Anders, Andreas Dischinger, Nils Ellmenreich, Peter Faber, Robert Günz, Harald Keimer, Radko Kubias, Wolfgang Meisl, Martina Schumergruber, Sabine Wetzel, Christian Wieninger and Alexander Wüst.*

Thanks to Jean-Francois Collard for several discussions, explanations, for proofreading and for offering his flat during a very inspiring stay in Paris which was made possible by a DAAD PROCOPE project.

I am very grateful to the members of the Lehrstuhl für Programmierung who often allowed me to use their desks and computers while they were on a business trip or somewhere else. I greatly appreciate especially the "long-term asylum" which Ulrike Lechner granted me.

Thanks to my friends in Gern — for not understanding exactly what I was working on, thus, for getting my mind off work, especially during the pre-lunch drink on Sundays.

Finally I would like to thank my parents: for making my study possible, for all kinds of support and for never putting me under pressure.

Contents

Chapter 0

Preface

This diploma thesis is part of a research project on building automatic parallelizers based on the polytope model. It examines methods for deriving parallel target programs that contain statements with execution spaces that are known only at run time. We call such spaces *dynamic execution spaces* (Chapter 2), in contrast to *static execution spaces* whose shape and width are entirely known at compile time. Dynamic execution spaces typically arise from source programs containing while loops. But there are also special kinds of for loops causing dynamic execution spaces. Their number of iterations (usually) cannot be predicted at compile time.
We will examine several classes of loops with their resulting spaces and discuss their requirements for parallelization.
Parts of the results are implemented as components of the automatic parallelizer for imperative programs, LooPo [13], under development at the Universität Passau, Germany.
Figure 0.1 shows the five main parts of LooPo that have to be carried out in sequence. Note that we depict scheduler and allocator as one module, because together they yield the transformation function that maps the source index space to the target index space.
We utilize the *polytope model* as a mathematical geometrical foundation. Its roots reach back to the late Sixties [15] and it was discovered for parallelizing compilers by Leslie Lamport in 1974 [18].
Parallelization in this setting proceeds along the following steps (refer to Figure 0.1):

- The *Parser* checks the syntactical correctness of the source program and computes the parse tree [14].

- During the *Dependence Analysis* the memory accesses are examined and the data dependences between operations as well as the index spaces of

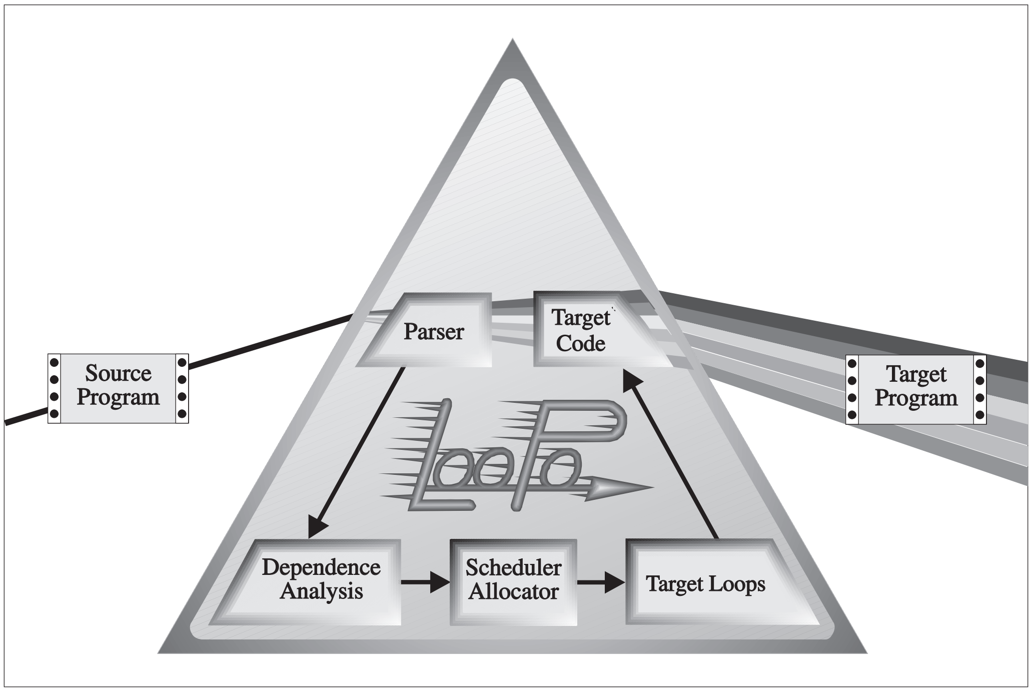

Figure 0.1: Structure of LooPo

the statements in the source program are calculated [16, 17]. These are modeled as polytopes resulting from intersections of half spaces given by the loop bounds. To be able to do so, the source program must only contain special kinds of loops, which we will discuss later.

- *Scheduler and Allocator* yield a function, the *transformation*, that tells us which operation is to be executed on which processor at what time [26, 21]. This transformation consists of a time component determined by the scheduler and a space component determined by the allocator. It is often also called the *space-time mapping*.
- *TargetLoops* takes the source polytopes and the transformation, calculates the target polytopes and transforms these target polytopes back to a nest of space and time loops [25]. We call this module also the *target generator*.
- The module *TargetCode* transforms the internal representation delivered by the target generator to real parallel programs coded in different parallel programming languages [5].

The polytope model turned out to be very useful for expressing and parallelizing loop programs under certain circumstances:

- The space-time mapping has to be a bijective affine function in loop indices and constant parameters.
- The lower *and* upper bounds of a loop have to be affine expressions in indices of enclosing loops and constant parameters. We describe ways of relaxing this restriction in this thesis.

There are other restrictions, e.g., the form of the dependences, which are not significant for this paper. We omit them.
The properties described above have some incisive implications:

- The extent of a statement's execution space is entirely known (parameterized) in *all* dimensions at compile time. This means we have only static source execution spaces.
- The shape of source execution spaces is a polytope, i.e., it has straight edges and is finite.
- We have a static target execution space whose shape is a polytope, too.
- The sequence of target loops is arbitrary since the target polytope can be expressed as a system of linear inequalities that can be solved in any order. In particular, the time loops can be outermost to get a synchronous target program or they can be innermost to achieve asynchronous parallelism. In contrast to the target program, the polytope model does not distinguish between space and time dimensions.
- Existing methods based on the polytope model cannot deal with dynamic execution spaces, whose shape and width are not known at compile time. Where we know the end of a for loop before the first operation of a statement within its body starts, the execution spaces containing while loops cannot even be supposed to be finite ex ante (at compile time), but – of course – should turn out to be at run time.

To solve the problems mentioned in the last item, an extension of the polytope model is necessary, the *polyhedron model* [10]. Dynamic execution spaces are modeled as polyhedra (which are infinite in some dimensions).
Our thesis examines ways of implementing the polyhedron model. This implementation should require as few changes of existing methods (that are shown in Figure 0.1) as possible.
Figure 0.2 shows the logical integration of the results of this paper into the flow of parallelization steps. They also could be implemented as an integral component of the respective module to let it appear more as a uniform whole.

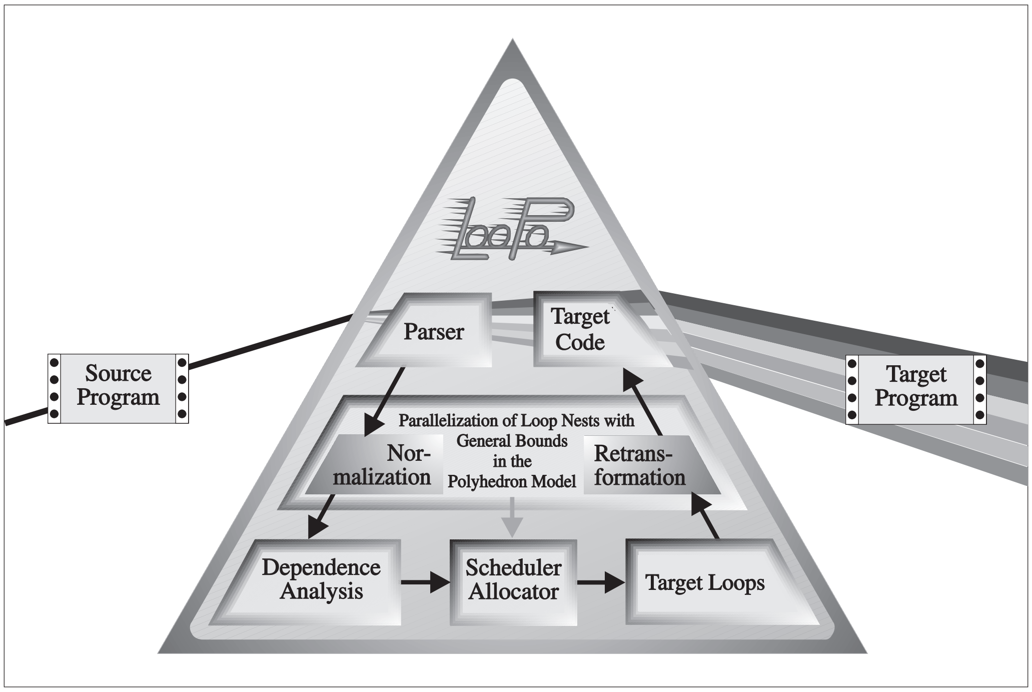

Figure 0.2: Structure of LooPo for handling dynamic execution spaces

Normalization and *Retransformation* are extensions to the parser and the target generator, the grey arrow symbolizes requirements which scheduler and allocator have to meet. More on this in Chapter 3.

Chapter 1 gives some basic definitions of concepts related to the polyhedron model. In Chapter 2 we consider several types of execution spaces and discuss their effects on parallelization. We will see that there is a great variety of execution spaces which cannot yet be treated by the existing methods in the polytope model. A theory for dealing with these (dynamic) execution spaces is presented in Chapter 3.

Chapter 4 describes how the theoretical methods introduced in Chapter 3 can be realized in the setting of LooPo. The major concepts here, as shown in Figure 0.2, are *loop normalization* and *retransformation*. The latter is necessary to restore the original execution spaces that are changed during normalization.

The last chapter gives a conclusion and some prospects on possible optimization and future work.

Chapter 1

Basic Definitions

This chapter gives an overview of some necessary, basic concepts and notation that we will use throughout this thesis. We are proceeding on the assumption that the reader is familiar with the concepts of linear algebra, particularly with matrices and their inverses, affine transformations, (systems of) linear equations and inequalities. We also assume that the principle of parallelization in the polytope model is known.

1.1 Mathematical Foundations

Definition 1 (Declaration of Functions).
Usually we declare functions in the following way:

$$f : D \rightarrow R : x \mapsto f(x)$$

where D is the domain and R is the range of the function with name f. $x \in D$ is a value to which f is applied and $f(x)$ is an arithmetic expression that evaluates to the value ($\in R$) of f when applied to x.

Our notation of quantifications and logical deductions follows Dijkstra [3]:

Definition 2 (Quantification).
Quantification over a variable x is denoted as follows:

$$(Q\ x\ :\ R(x)\ :\ P(x))$$

where $Q \in \{\forall, \exists\}$ is a quantifier, R is a predicate that determines the range of the values of x and P is any predicate depending on the values of x.

Definition 3 (Formal Logical Deduction).
We denote formal logical deductions in the form:

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

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

Definition 4 (Componentwise Partial Order on Vectors).
Let $\vec{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_r \end{pmatrix}$ and $\vec{x}' = \begin{pmatrix} x'_1 \\ \vdots \\ x'_r \end{pmatrix}$ be two r-dimensional vectors. Then we define the componentwise orders $<$ and $\leq$ on vectors as follows:

$$\begin{array}{lcl} \vec{x} < \vec{x}' & \Leftrightarrow & (\forall\, k \,:\, 1 \leq k \leq r \,:\, x_k < x'_k) \\ \vec{x} \leq \vec{x}' & \Leftrightarrow & (\forall\, k \,:\, 1 \leq k \leq r \,:\, x_k \leq x'_k) \end{array}$$

'$>$' and '$\geq$' are defined analogously.

Remark. We use the same componentwise order analogously for r-tuples and row vectors.

Let us now recall some mathematical concepts that are fundamental for our topic.

Definition 5 (Halfspace).
Let $a_1, \ldots, a_n, b \in \mathbb{R}$, $n \in \mathbb{N}$. Then a *halfspace* $\mathcal{H}$ of $\mathbb{R}^n$ is defined as

$$\mathcal{H} = \{(x_1, \ldots, x_n) \in \mathbb{R}^n \mid a_1x_1 + \cdots + a_nx_n \leq b\}$$

Definition 6 (Polyhedron).
An n-dimensional *polyhedron* in $\mathbb{R}^n$ is the intersection of a finite number of halfspaces of $\mathbb{R}^n$.

Definition 7 (Polytope).
An n-dimensional *polytope* in $\mathbb{R}^n$ is a bounded n-dimensional polyhedron in $\mathbb{R}^n$.

Definition 8 (Convex Set).
Let $a \in \mathbb{R}$, $a \in [0, 1]$. A set $\mathcal{S} \subseteq \mathbb{R}^n$, $n \in \mathbb{N}$ is *convex*, iff:

$$(\forall\, x, y \,:\, x, y \in \mathcal{S} \,:\, (ax + (1-a)y) \in \mathcal{S})$$

Remark. Every intersection of a finite number of halfspaces is convex. Thus, every polyhedron and every polytope is convex, too.

The definitions above are all given with respect to $\mathbb{R}$. However, we are only interested in those points x whose coordinates are integer values, i.e., $x_k \in \mathbb{Z}$, $1 \leq k \leq n$. Only these points are considered, because the code generation techniques available to us [25] can only process such index spaces (see also Definition 16).

1.2 Legal Input Programs

From the point of view of this paper, a legal source program may be composed of elements described in this section.

Definition 9 (Identifier).
There are three possible kinds of identifiers:

- *Loop indices* or *loop variables* run from the lower bound of a loop to the upper bound and are incremented by the stride (see also Definitions 11 and 12). A loop index is only 'visible' in the body of its respective loop but may not be changed by any other statement except for the loop statement itself (see also Figure 2.2 on Page 23).
- *Variables* may be changed arbitrarily, in contrast to loop indices. Note that our method does not exclusively require arrays (possibly 0-dimensional), however, the capabilities of LooPo are restricted to arrays — so far.
- *Structure parameters* represent the problem size. They are initialized during the loading of the program and are *only read* during run time, i.e., they are constant throughout the whole execution of the program. We will also call them *constants*.

Definition 10 (if Statement).
We allow plain if statements of the form:

if *condition* then *body* endif

with all kinds of conditions.

Remark. Although we do not explicitly consider if statements with an else branch in this thesis, this implies no loss of generality. Every if statement with an else branch can be split into two plain if statements: the then branch keeps its original condition and the else branch becomes an if statement with the negation of the original condition.

Definition 11 (for Loop).
Allowed are for loops with all kinds of loop bounds. For technical reasons we only allow integer values ($\in \mathbb{Z}$) for loop indices, although the theory would permit rational values ($\in \mathbb{Q}$). The stride has to be composed of integer structure parameters. Thus, it is decidable before run time whether the stride is positive or negative and this does not change during run time. Notation:

$$\mathsf{for}\ index\ := lower\_bound\ \mathsf{to}\ upper\_bound\ \mathsf{step}\ stride\ \mathsf{do}\ body\ \mathsf{end}$$

We call *lower_bound* and *upper_bound* the *bound expressions.*
If the stride is not given explicitly, we assume it to be 1.
The values of *lower_bound*, *upper_bound* and *stride* are evaluated before the execution of the loop. If we reference these fixed values, we use the symbols *LB*, *UB* and *ST*, respectively.

Definition 12 (while Loop).
A while loop is usually denoted as follows:

$$\mathsf{while}\ condition\ \mathsf{do}\ body\ \mathsf{end}$$

As we have to specify the index space of a body statement of a while loop, we view while loops — according to [10] — as generalized for loops and provide them with a new index:

$$\mathsf{for}\ newindex := 0\ \mathsf{while}\ condition\ \mathsf{step}\ stride\ \mathsf{do}\ body\ \mathsf{end}$$

newindex is just a counter for the number of iterations of the respective while loop. Therefore the *stride* usually (and in our thesis always) is set to 1. However, we could permit the same types of strides as are permitted for for loops.
With "upper bound of a while loop" we mean the number of iterations the loop actually carries out. This value is only known after the loop's termination and therefore is not explicit in the syntax of a while loop.
With this notation we have an explicit lower bound for the respective dimension of the execution space and the loop condition, which describes the upper bound of this dimension in an implicit way.

body is the set of statements whose execution is repeated (in case of a loop body) or whose execution depends on the truth value of the if condition (in case of an if body).
Note that one and the same statement can belong to more than one body: it belongs to all bodies of its enclosing loops and if statements.

Remark. The notation introduced in Definition 12 can easily be translated to the usual syntax:

```
newindex := 0
while condition do
   body
   newindex := newindex + stride
end
```

The 'new' syntax makes the role of the new counter explicit (see also Figure 2.4 on Page 26) and is therefore more expressive. *newindex* is treated as it were the index variable of a usual for loop.

Definition 13 (Statement).
In our context a statement may be an if statement or any conventional statement, usually an assignment, in the source program.

In Chapter 2 we split the head of a for or while loop into several statements, e.g., the evaluation of the lower bound (see Figures 2.2 and 2.4). To make clear that we do not view the head of a loop as one single statement, we call it for *instruction*, while *instruction* or just *loop instruction* throughout this thesis.
In most cases a statement is an assignment and is uniquely defined by its left side. For instance, let $x := y$ a statement. Then we call this statement "statement x" or "x statement".

Definition 14 (Some Sets).

- The symbol for the *set of statements* in a source program $\mathcal{P}$ is: $\mathcal{S}_{\mathcal{P}}$.
- The symbol for the *set of index variables* in a source program $\mathcal{P}$ is: $\mathcal{L}_{\mathcal{P}}$.
- $\mathcal{E}_s$ denotes the *set of enclosing loop indices* of a statement s, i.e., it contains all indices of loops whose bodies contain s.

  The set of enclosing loop indices also identifies the *loop nest* of a statement s. This loop nest determines the index space of s (see Definition 16).

A loop nest is said to be *perfect* if, for all statements, the sets of enclosing loop indices are equal.

Definition 15 (Dimensionality or Level of a Statement).
Let $s \in \mathcal{S}_{\mathcal{P}}$ be a statement in program $\mathcal{P}$. The value of the function

$$dim : \mathcal{S}_{\mathcal{P}} \to \mathbb{N} : s \mapsto |\mathcal{E}_s|$$

is called the *dimensionality* or *level* of s.

We can associate the level of any a loop instruction with its respective loop index and use this numbering as a basis for defining an ("outside to inside") order on the indices of enclosing loops of a statement.

Remark. A loop is identified by its loop instruction. If the dimensionality of the loop instruction is r than we often speak of "a loop at level r".

1.3 Spaces and their Transformation

In this section we describe the essential concepts of the model we use to represent the loop nests of our source programs. The index and execution spaces are the interfaces between the programs and the mathematical model.

Definition 16 (Index Space).
Let $s \in \mathcal{S}_{\mathcal{P}}$ be a statement in the source program $\mathcal{P}$, and d_s the dimensionality of s. Further let i_r be the value of the loop index of a loop surrounding s at level r. Then the *index space of statement* s is defined as a subset of $\mathbb{Z}^{d_s}$:

$$\mathcal{I}_s := \{(i_1, \ldots, i_{d_s}) \in \mathbb{Z}^{d_s} \mid (\forall\, r \,:\, 1 \le r \le d_s \,:\, (i_r - LB_r)\%ST_r = 0 \;\wedge \\ \left\langle \begin{array}{ll} LB_r \le i_r \le UB_r & \text{if } i_r \text{ belongs to a } \textsf{for} \text{ loop} \\ LB_r \le i_r & \text{if } i_r \text{ belongs to a } \textsf{while} \text{ loop} \end{array} \right\rangle)\}$$

In the programming language C '%' is the sign for the modulo operator. We use this notation throughout this thesis.

Remark. Let $x = (i_1, \ldots, i_{d_s}) \in \mathcal{I}_s$. Every position r in the tuple, $r \in \{1, \ldots, d_s\}$, uniquely identifies the loop index of a surrounding loop at level r. To express this, we use the function $indexof_s : \{1, \ldots, d_s\} \to \mathcal{E}_s$. We will also need the three additional predicates $is\_for(r)$, $is\_naf(r)$ and $is\_whl(r)$ that indicate whether position r identifies a for loop (any type), a non-affine for loop whose bounds are only non-affine expressions in loop indices and structure parameters, or a while loop.

Definition 17 (Operation, Instance of a Loop).
One *operation* is an instance of a statement. A statement in a loop body is executed several times with different values for the indices of the surrounding loops. In the program it occurs as one statement; during the execution this single statement appears as several operations.
Our notation of an operation is $\langle s, x_s \rangle$, where $s \in \mathcal{S}_\mathcal{P}$ and $x_s \in \mathcal{I}_s$.

We call the loop identified by an operation of a loop instruction an *instance of the loop*, given by its loop instruction and the values of the indices of enclosing loops.

Remark. The substatements of loop instructions have operations, too. Thus, the value of the lower bound of a loop at level k, lb_k, is a function of the indices of the surrounding loops[1], $lb_k(i_1, \ldots, i_{k-1})$. We will use this notation only if this aspect is of interest in a given context. Otherwise we will simply write lb_k. The same applies for the upper bounds.

The index space of a statement represents the set of possible operations of this statement.

If some dimension of the index space is determined by a while loop then we have an infinite index space. Thus, there is a difference between the points in the index space and the points actually executed, as while loops are supposed to terminate and not to enumerate infinitely many points.
A similar situation arises when we have an if statement around a statement s. Then there are also points in the index space that do not contribute to an operation of s that is actually executed (see Definition 17).

Example 1.
Consider the following program:

```
for i := 0 to 5 do
   if i <> 3 then
      s
   endif
end
```

The index space of s is: $\mathcal{I}_s = \{i \in \mathbb{Z} \mid 0 \leq i \leq 5\} = \{0, 1, 2, 3, 4, 5\}$. But now there is a point $3 \in \mathcal{I}_s$ whose respective operation $\langle s, 3 \rangle$ is not executed by the program.

[1]It is also a function of other variables occurring in the bound expression, but the values of these variables also depend on the iteration. So for the loop bounds it is sufficient to consider only the indices of the surrounding loops.

To distinguish these two different sets of points, we need the concept of the *execution space* [11]. At this stage, we only give an informal definition and leave the formalization up to Chapter 3.

Definition 18 (Source Execution Space).
Let $s \in \mathcal{S}_\mathcal{P}$ be a statement in the source program $\mathcal{P}$. Then the *source execution space of statement* s, $\mathcal{X}_s$, is defined as a subset of $\mathcal{I}_s$. It contains all points at which an operation of s is actually carried out at run time.
The source execution space of the entire program is $\mathcal{X} = \bigcup_{s \in \mathcal{S}_\mathcal{P}} \mathcal{X}_s$.

Remark. For a statement that is not in the body of a loop nest with while loops and that is not guarded by if statements, the source index space is equal to the source execution space, $\mathcal{I}_s = \mathcal{X}_s$. Statements that belong to the same body and have the same dimension have the same index and execution spaces.

In Chapter 3 the difference between index and execution space will become important: we will have to change the original index space of a statement but have to take care not to change its respective execution space.

Another important concept for parallelization is that of the dependences between operations. They express a necessary execution ordering between the operations involved.
There are several special kinds of dependences, but for our needs a general definition is sufficient [26].

Definition 19 (Dependence).
Let $s, s' \in \mathcal{S}_\mathcal{P}$ be two (not necessarily different) statements in program $\mathcal{P}$ and let $\langle s, x_s \rangle$ and $\langle s', x_{s'} \rangle$ be operations of s and s' with $x_s \in \mathcal{X}_s$ and $x_{s'} \in \mathcal{X}_{s'}$.

- *Data dependence*: operation $\langle s', x_{s'} \rangle$ is *data dependent* on operation $\langle s, x_s \rangle$, denoted $\langle s, x_s \rangle \delta^d \langle s', x_{s'} \rangle$, if:
  1. $\langle s, x_s \rangle$ and $\langle s', x_{s'} \rangle$ access the same memory cell and at least one of them is a write access, and
  2. $\langle s, x_s \rangle$ is executed before $\langle s', x_{s'} \rangle$ in $\mathcal{P}$.
- *Control dependence*: operation $\langle s', x_{s'} \rangle$ is *control dependent* on operation $\langle s, x_s \rangle$, denoted $\langle s, x_s \rangle \delta^c \langle s', x_{s'} \rangle$, if:
  1. $\langle s, x_s \rangle$ evaluates a predicate and
  2. the execution of $\langle s', x_{s'} \rangle$ depends on the value of this predicate.

s is called the source, s' the target of the dependence.
Both types of dependences impose a temporal order on the involved operations. If the difference between data and control dependence does not matter in a given context, we omit the superscripts c and d of δ.
Sometimes it is sufficient to know which statements are dependent on each other. In this case we write $s\delta s'$ instead of $\langle s, x_s\rangle\delta\langle s', x_{s'}\rangle$. Again the refinements δ^d and δ^c are possible.

Remark. A structure parameter is initialized before the actual execution of the program starts. During run time these values are only read. Thus, according to Definition 19, structure parameters do not cause any data dependences.

Definition 20 (h-Transformation).
Let $\langle s, x_s\rangle\delta\langle s', x_{s'}\rangle$. Then δ defines a relation between the execution spaces of s' and s. Because up to now only affine dependences can be handled and the source of a dependence is unique, this relation can be expressed as an affine function, the *h-transformation* [7]:

$$h \;:\; \mathcal{X}_{s'} \to \mathcal{X}_s \;:\; x_{s'} \mapsto x_s$$

Remark. Affine functions can be represented as a matrix. In the case of an h-transformation, this is a $(d_s{\times}d_{s'}{+}1)$-matrix whose columns correspond to the dimensions of statement s' plus the constant portion of the affine function. The rows correspond to the dimensions of s.

Based on the dependences, each operation is mapped to a special point in time at which it is to be executed. This time mapping is called the *schedule*. To be able to express the requirements a valid schedule must meet, we need the lexicographical order on an r-dimensional vector space $\prec$. Sometimes we do not need the strict order and use $\preceq$ to denote this.
Note that $\prec$ and $\preceq$ are different from the $<$ and $\leq$ operators on vectors (see Definition 4).

Definition 21 (Schedule).
The function $\tau_s : \mathcal{I}_s \to \mathbb{Z}^{d_s^{\mathrm{t}}}$, $d_s^{\mathrm{t}} \in \mathbb{N}$ is called a d_s^{t}-dimensional *schedule for statement* s (superscript t stands for *time*), if it preserves the dependences between all operations of s and all operations of any other statement in $\mathcal{P}$, i.e.,

$$(\forall\, x_s, x_{s'} \;:\; x_s \in \mathcal{I}_s \;\;\wedge\;\; x_{s'} \in \mathcal{I}_{s'} \;:\; \langle s, x_s\rangle\delta\langle s', x_{s'}\rangle \;\Rightarrow\; \tau_s(x_s) \prec \tau_{s'}(x_{s'}))$$

Remark. In our setting the schedule is always an affine function, so it can be expressed by a $(d_s^{\mathrm{t}} \times d_s + 1)$-matrix whose columns correspond to the indices of the surrounding loops and whose rows reflect the various dimensions of the schedule. The last column represents the constant portion of the affine function.

τ_s assigns every operation of s to a certain instant of time, at which it is to be executed.

Definition 22 (Allocation).
The function $\alpha_s : \mathcal{I}_s \to \mathbb{Z}^{d_s^{\mathrm{p}}}$, $d_s^{\mathrm{p}} \in \mathbb{N}$, defines the place (the virtual processor coordinates) at which the operations of s are to be carried out.

Just as a schedule, an allocation can also be expressed by a matrix, a $(d_s^{\mathrm{p}} \times d_s + 1)$-matrix.

Definition 23 (Transformation, Transformation Matrix).
The *transformation of a statement* s is defined by both, the schedule *and* the allocation of this statement:

$$T_s : \mathcal{I}_s \to \mathbb{Z}^{d'_s} : x_s \mapsto (\tau_s(x_s), \alpha_s(x_s)) \text{ , where } d'_s = d_s^{\mathrm{t}} + d_s^{\mathrm{p}}$$

To get the *transformation matrix for a statement* s, we compose the matrices for schedule and allocation to a single $(d'_s \times d_s + 1)$-matrix by positioning the matrix for the allocation below the matrix for the schedule.

Remark. Schedule and allocation are often calculated separately. Thus, in general, the transformation will not be bijective. Wetzel [25], however, presents methods for dealing with non-bijective and non-unimodular transformations. Consequently we may (and do) assume to have bijective unimodular transformations only, as this is no limitation to generality.

Definition 24 (Target Space).
We define the *target space* or *transformed index space* of statement s as a subset of $\mathbb{Z}^{d'_s}$:

$$\mathcal{TI}_s = \{x'_s \in \mathbb{Z}^{d'_s} \mid (\exists\, x_s : x_s \in \mathcal{I}_s : x'_s = T_s(x_s))\}$$

d'_s is the dimension of the target space for statement s.

Analogously to the source execution space we also have a target execution space of a statement.

Definition 25 (Target Execution Space).
The *target execution space* or *transformed execution space* of a statement s is defined as a subset of its target space:

$$\mathcal{TX}_s = \{x'_s \in \mathcal{TI}_s \mid (\exists\, x_s : x_s \in \mathcal{X}_s : x'_s = T_s(x_s))\}$$

In contrast to the target execution space we will often call the source execution space just *execution space*.

Remark. While different statements in the same body have the same index (execution) spaces, in general, they have different target (execution) spaces. The reason for this is that every statement can have a different transformation.

Definition 26 (Scanning and Scannability).
The enumeration of (source and target) execution spaces by a nest of loops is called *scanning*.
An execution space is *scannable* iff it can be scanned precisely by a nest of loops.

Remark. Scannability holds trivially for source execution spaces, because they are given by loop nests [9]. So the property of scannability is mainly interesting for target execution spaces.

On closer inspection we notice that the scannability property of target execution spaces depends on the transformation and on the shape of the respective source execution space, as these two determine the appearance of a target execution space.
In [10] the predicate *scannability* is defined for space-time mappings. It ensures that the target execution space is scannable, regardless of the shape of the source execution space.
We will have a closer look at the implications of scannability in Chapter 2.
With the presence of general loop nests, the transformed execution space $\mathcal{TX}_s$ of a statement s may be unscannable [11]. In this case it is not possible to find a set of nested loops that enumerates the target execution space precisely.

Definition 27 (Scanned Spaces).
The set of actually scanned source (target) points of a statement s, the *scanned source (target) space* $\mathcal{S}_s$ ($\mathcal{TS}_s$), is defined as the set of points that are enumerated by the source (target) program.

Remark. If we have if statements in the source program, then — in general — the scanned source space of a statement in the body is different from its execution space.

We will have to prove that our implementation ensures that the target program scans a superset of the transformed execution space, i.e., that $\mathcal{TX}_s \subseteq \mathcal{TS}_s$ for all $s \in \mathcal{S}_\mathcal{P}$. Further, we must prevent an operation from being executed at a point that does not belong to the transformed execution space of its respective statement.

1.4 Summary of Frequently Used Symbols

Let us summarize the symbols and notation introduced in this chapter, as these will often recur throughout this thesis:

Programs:

$\mathcal{P}$	:	identifier for a (source) program
s	:	identifier for a statement
$\mathcal{S}_\mathcal{P}$	:	set of statements occurring in a program $\mathcal{P}$
$\mathcal{L}_\mathcal{P}$	:	set of all loop indices occurring in program $\mathcal{P}$
$\mathcal{E}_s$	:	set of indices of enclosing loops of a statement s

Loops:

LB_r	:	fixed value of the lower bound of a loop at level k
UB_r	:	fixed value of the upper bound of a loop at level k
ST_r	:	value of the stride of a loop at level r
i_r	:	value of the index variable (loop index) of a loop at level k,
$cond_k$	:	boolean value, usually the value of the condition of a while loop or an if statement
$body_r$	:	body of a loop at level r

Source spaces:

r	:	in the current context a fixed bound, but variable in general, often $1 \leq r \leq d_s$
k	:	index, often $1 \leq k \leq r$
d_s	:	(source) dimension of a statement s
$\mathcal{I}_s$	:	index space of statement s
$\mathcal{X}_s$	:	execution space of statement s
x_s	:	point in the source space $\mathcal{I}_s$ of statement s, usually $x_s = (i_1, \ldots, i_{d_s})$
$\langle s, x_s \rangle$	:	operation of statement s

Transformation:

δ^d	:	data dependence
δ^c	:	control dependence
δ	:	dependence
$\prec, \preceq$	:	lexicographical order (used to describe a temporal order between operations)
τ_s	:	schedule of statement s
α_s	:	allocation of statement s
T_s	:	transformation of statement s

Target spaces:

d_s^{t}	:	dimension of the schedule of statement s
d_s^{p}	:	dimension of the allocation of statement s
d'_s	:	target dimension of a statement s
$\mathcal{TI}_s$	:	target index space of statement s
$\mathcal{TX}_s$	:	target execution space of statement s
$\mathcal{TS}_s$	:	scanned target space of statement s
x'_s	:	point in the target space $\mathcal{TI}_s$ of statement s, usually $x'_s = (i'_1, \ldots, i'_{d'_s})$

Remark. We will omit subscripts if the meaning is clear without them in a given context.

Chapter 2

Classification of Execution Spaces

An important step of a parallelization in the polyhedron model is the determination of the target execution spaces and the identification of a nest of target loops (space and time loops) which enumerates them. One main contribution of this thesis is an examination of the correlation between different types of execution spaces and their images under certain transformations.
Note that we restrict ourselves to piecewise affine transformations, since our schedulers and allocators only yield piecewise affine space-time mappings.
To show the differences between the various target execution spaces, we introduce a classification of source loop types affecting the respective dimension of the execution space. The examination of the execution spaces and transformations can be performed with varying precision: a more precise examination means on the one hand more effort but on the other hand it reveals refined classes for which better methods of parallelization can be found.
Figure 2.1 shows the different classes we shall define in this chapter. In general, from left to right the potential for parallelism decreases and the run time overhead increases. From the top to the bottom the effort of analyzing the execution spaces (at compile time) increases and the classes get more precise. The target execution spaces of the classes in the light grey boxes can theoretically be scanned precisely. For all other classes, in general, we have to scan a superset of the target execution space and watch out for operations which must not be carried out at some iteration.

Remark. Our classification is only based on instructions that constitute index spaces, namely on `for` and `while` loops. Thus, it does not consider `if` statements. We make remarks on the consequences of `if` statements at the appropriate points in the following sections.

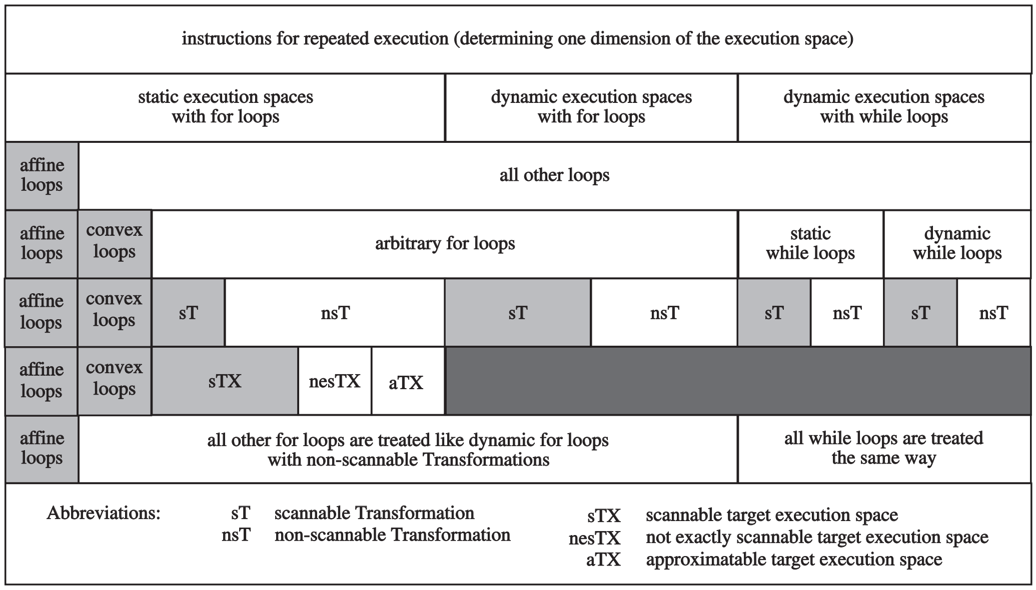

Figure 2.1: Different classes of execution spaces

Our aim is to learn as much as possible about the target execution spaces at compile time. If we knew them exactly, we could enumerate them in parallel without any run time overhead. In this case the only sequentiality arises from data dependences between operations of statements in the body — not from calculating the execution spaces themselves.
We can use the types of loops in the source program, their bounds and the transformation function to describe the target execution space. The more we can take into account the better our results will be.

Remark. In this thesis we are only interested in scanning (a superset of) the target execution space and filtering the points that must not be executed, because their inverse image does not belong to the execution space. We consider only dependences that have to do with these problems and do not take into account further dependences that arise from the program itself. It is up to the programmer to invent algorithms that impose as few data dependences as possible or to apply other tools (e.g., [17]) that eliminate some unnecessary data dependences.

In the following sections we will discuss the classes shown in Figure 2.1 and present some examples for illustration. The maximum dimensionality of the execution spaces of statements in the examples is 2. We use the third dimension for depicting operations of different statements; operations with the same third coordinate belong to the same statement. If we use transformation matrices, the first line corresponds to the schedule (t) and the second line to the allocation (p). In the examples we consider only target programs whose outermost loops are time loops (synchronous target programs).

2.1 A Glance at Sequential Execution Spaces

Every for loop and while loop[2] determines one dimension of the execution spaces of the statements in its body. All execution spaces are scanned in sequence and at every point the operations are ordered according to the textual order of their statements in the source program.
In the next two subsections we take a closer look at for and while instructions. This may seem somewhat trivial, and usually we do not think about their logical structure when we use sequential for loops or while loops, but it is essential for the understanding of why and how loop nests can be parallelized.

2.1.1 Sequential for Loops

The nature of a single for loop is that the bounds and the stride are evaluated first and do *not change* during the execution of the body[3].
The values of the loop index of a for loop can be described by the following condition.

Definition 28 (Execution Condition of a for Loop).
The *execution condition of a* for *loop* is given by the (usual) semantics of for loops and is defined as follows:

$$ex\_condf_r(i_1, \ldots, i_r) :\equiv LB_r \leq i_r \leq UB_r \quad \wedge \quad (i_r - LB_r)\% ST_r = 0$$

The body is executed for all i_r which satisfy the execution condition. In all other cases the loop terminates.

The expressions for LB_r and UB_r are evaluated once before the respective body is executed. Only the value i_r is changing during the execution of the loop: after each iteration the stride ST_r is added to i_r.

[2] We do not consider repeat loops, they can be easily transformed to while loops.

[3] The programming language C has a different view of for loops: the bounds and the stride of a for loop are evaluated before each new iteration. If, e.g., a variable occurring in a bound expression has changed during the last iteration, this affects the value of the bound for the next iteration. In C every while loop can be denoted as a for loop.

As the index increases monotonously and all other values in the bound expressions do not change, the number of iterations is known at the beginning of the loop's execution, i.e., the extent of an instance of such a loop is known before the first operation of a statement in its body is executed.

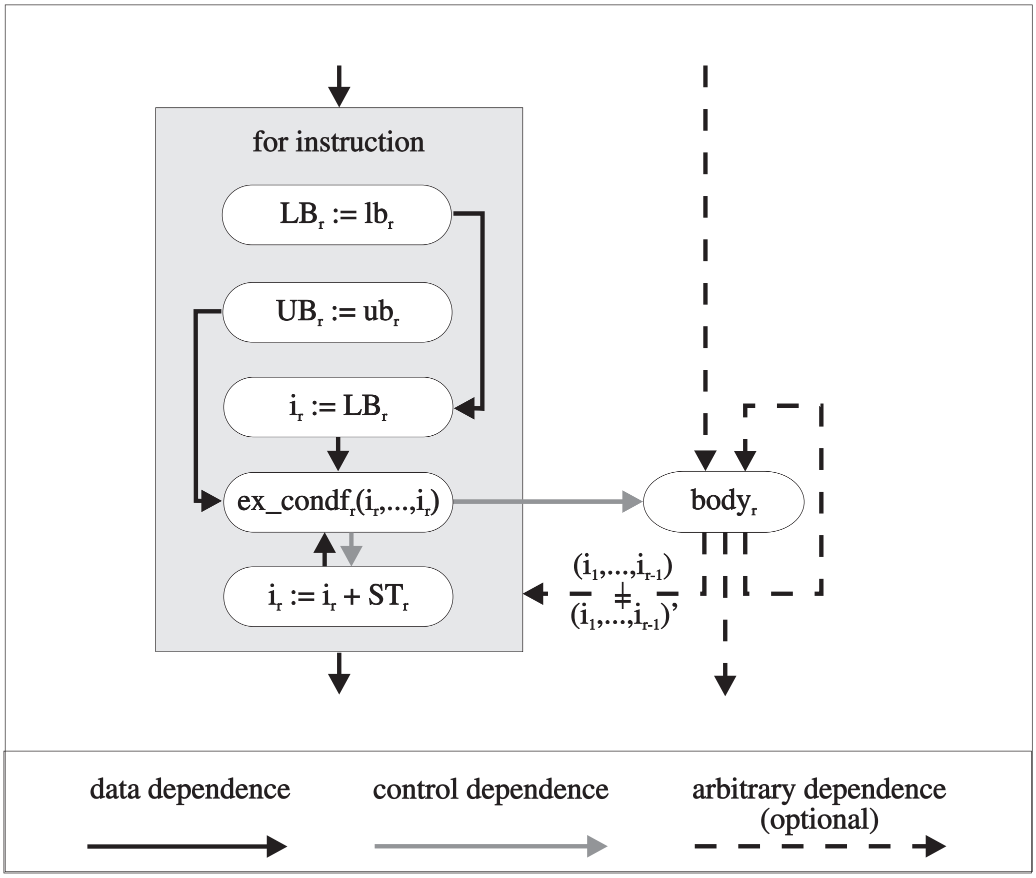

Figure 2.2: Dependences related to a for-instruction

Figure 2.2[4] shows the dependences caused and made possible by a for instruction in a sequential program. The grey rectangle contains all information given by the syntax of a for loop. The two assignments $LB_r := lb_r$ and $UB_r := ub_r$ are used to symbolize that the values of the bounds are fixed before the loop starts. We will call these statements *lower bound assignment*, *upper bound assignment* or *bound assignment* if we mean one of them. From now on we will denote them by lb_ass_r, ub_ass_r and bd_ass_r, respectively, if

[4]The way we show the dependences related to loop instructions and if statements are inspired by [28].

they belong to a loop instruction at level r.
The dotted dependences attached to the border of the rectangle can only be caused by the expressions for lb_r and ub_r.
The grey arrows represent control dependences to the body and to the induction (see [22]) whose execution depends on the boolean value of the execution condition: an operation of a body statement is only executed if the execution condition evaluates to *tt*. Control dependences may coincide with data dependences in the same direction, i.e., the control dependences exist definitely and may make some similar data dependences redundant (we omitted the data dependences for the sake of clarity). This is always the case for the dependence of the induction on the condition, as i_r has to be read before it may be incremented if the condition yields *tt*.
We see that the body of an instance of a for loop that is currently being executed cannot affect the bounds of its own execution space, but obviously it can affect the bounds of, e.g., the next instance of the same loop or the bounds of an inner loop, if there is one. This is expressed by the adornment $(i_1, \ldots, i_{r-1}) \neq (i_1, \ldots, i_{r-1})'$ for the dependence from the body to the for instruction and by the possible dependences pointing to and away from the body.
This observation will become important for parallel execution.
To summarize: we know the extent of an instance of a for loop before the first operation of its body is carried out. This holds independently from the appearance of the bound expressions.

2.1.2 Sequential while Loops

In contrast to for loops, the number of iterations of a while loop is not known before the loop terminates and the value of the condition has to be checked at the beginning of every new iteration. The body of a while loop may — and, for getting non-trivial loops, must — change the values occurring in the condition. In general, not only one "well-chosen index" changes in a monotone manner, but any value may change and affect the result of the next evaluation of the while condition.
If we take a look at Figure 2.3, we notice that the syntax of a while loop provides much less information compared to that of a for loop.
In general, we do not know how the values occurring in the condition will change before its next evaluation. These changes (the induction [22]) take place somewhere in the body.
The initialization of the changing values is not given syntactically, either. We know only the condition and that there are control dependences. One is directed from the condition to the loop body and another one points to the

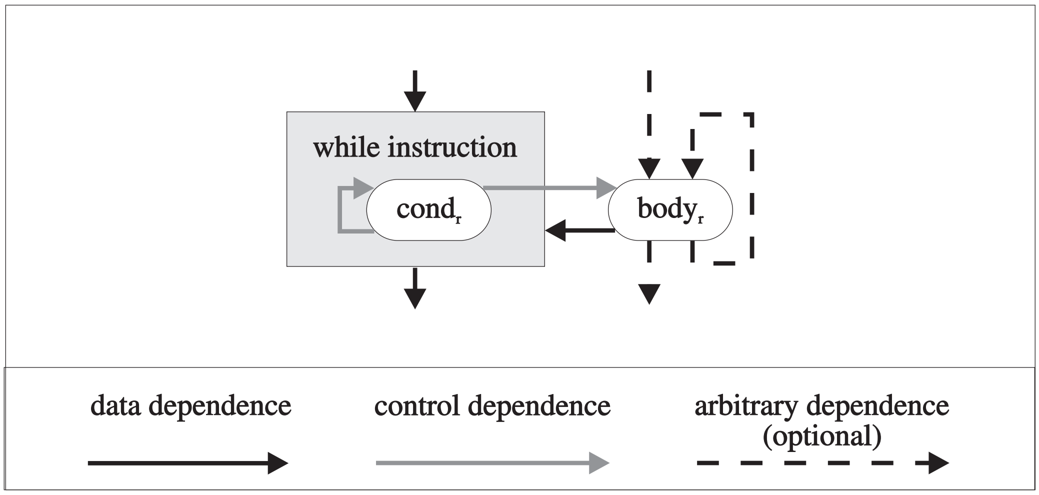

Figure 2.3: Dependences related to a while instruction

condition at the next iteration of the same loop instance. We call the latter one the while *dependence* [10].

In contrast to for loops, the dependence from the body to the while instruction is unlabeled. That means that it may be of an arbitrary form[5] and, especially, may point to the while instruction of the currently executed instance of the loop. Thus, a while loop can change the extent of its own execution space.

The facts just mentioned entail that we cannot predict the number of iterations a while loop will carry out. The loop's termination only becomes known when the condition evaluates to *ff* after some iteration.

However, Figure 2.3 only shows what is commonly related to a while loop. This view is sufficient for considering sequential loops, but for examining the parallel execution of while loops we have to take a closer look. In Figure 2.4 we present a more general view of while loops and show also the integration of the iteration counter (see Definition 12).

For while loops we can also define an execution condition, similarly to for loops. It is defined recursively according to the while dependence and therefore is more complicated to calculate.

[5] Here 'arbitrary' means: "all kinds of dependences possible in a sequential program".

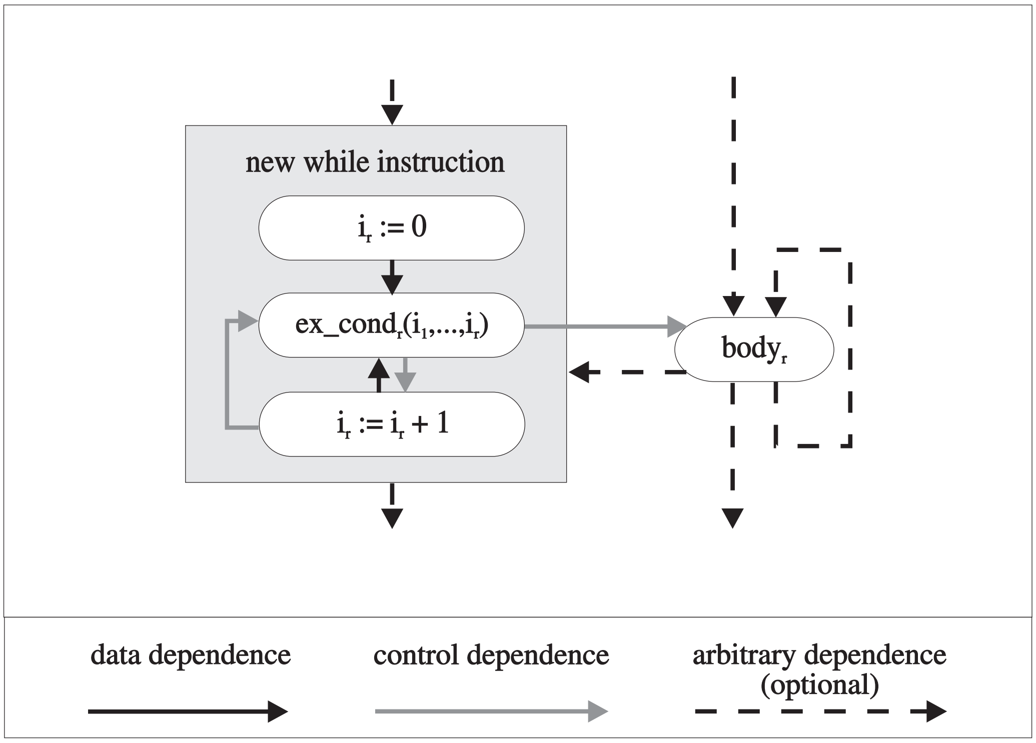

Figure 2.4: Dependences related to a while instruction given by the new syntax

Definition 29 (Execution Condition of a while Loop).

The *execution condition of a* while *loop* at level r is given by the semantics of while loops and is defined as follows:

$$
\begin{aligned}
&ex\_condw_r(i_1, \ldots, i_r) :\equiv \\
&\qquad\qquad (0 \leq i_r \wedge (i_r - 0)\%1 = 0) \;\; \wedge \\
&\qquad\qquad (0 = i_r \wedge cond_r(i_1, \ldots, i_r)) \;\; \vee \\
&\qquad\qquad (0 < i_r \wedge ex\_condw_r(i_1, \ldots, i_r - 1) \wedge cond_r(i_1, \ldots, i_r))
\end{aligned}
$$

The first line describes the index space of the loop. As in our thesis the lower bounds and the strides of all while loops are always 0 and 1, respectively, we could also omit this line. However, we want to remember the general case (arbitrary lower bounds and strides) and choose to keep it. The following two lines restrict the index space to the actual execution space of the loop: the first iteration of a while loop is carried out if the loop condition holds, the following iterations are executed if the previous one was executed and the condition holds. For $i_r < 0$ the execution condition yields *ff*.

The definition of ex_condw_r follows the definition of the predicate *executed* in [1]. However, we take only the parts that relate to one level of the loop nest, i.e., here we have a single-dimensional view. In Chapter 3 we combine the execution conditions of every dimension to one execution condition for every statement.
Now that we can express the values of the loop indices we can give a formal definition of an execution space.

Definition 30 (Source Execution Space).
Let $s \in \mathcal{S}_\mathcal{P}$ be a statement in a source program $\mathcal{P}$ that does not belong to the body of an if statement. Then the *source execution space of statement s* is defined as a subset of $\mathcal{I}_s$:

$$\begin{array}{c}\mathcal{X}_s = \{(i_1,\ldots,i_{d_s}) \in \mathcal{I}_s \mid (\forall\, k : 1 \le k \le d_s : \\ (ex\_condf_k(i_1,\ldots,i_k) \wedge is\_for(k)) \;\vee\; (ex\_condw_k(i_1,\ldots,i_k) \wedge is\_whl(k)))\}\end{array}$$

The source execution space of the entire program is $\mathcal{X} = \bigcup_{s\in\mathcal{S}_\mathcal{P}} \mathcal{X}_s$.

After this closer look at sequential loops we shall describe the loop classes. Each of the following sections corresponds (downward) to a row in Figure 2.1.

2.2 Static and Dynamic Execution Spaces

We distinguish two main kinds of execution spaces, determined by the dependences that point to the loop instructions.
The first one is called *static execution space*. Its shape and extent is entirely known at compile time or can at least be approximated by a superset whose shape and extent is known at compile time. Static execution spaces are characterized by loop bounds to which no data dependences are 'pointing'.

Definition 31 (Static Execution Space).
Let s and s' be two statements in program $\mathcal{P}$, where s' is not part of a loop instruction. The dimensionality of s is denoted as d_s. Let further bd_ass_r be a bound assignment of a loop surrounding s at level r.
We call the execution space $\mathcal{X}_s$ of s *static* if the following holds:

$$(\forall\, r : 1 \le r \le d_s : \neg(s' \, \delta^d \, bd\_ass_r))$$

These spaces could be scanned in one time step (i.e., in parallel) by forall loops. However, dependences between the body statements may spoil this parallel execution.

Remark. while loops can only belong to a static execution space if the values occurring in their conditions are not changed by any statement. This renders only while loops that have either no (the while condition always yields *ff*) or infinitely many iterations (the while condition always yields *tt*). It can be determined at compile time which case applies. So, usually execution spaces determined by one or more while loops are non-static (dynamic, see Definition 32).

Example 2 (Static Execution Space).
The loop nests in the following two programs yield the static execution spaces shown in Figures 2.5 and 2.6.
No loop bound depends on the computations of any other statement, so we know the execution spaces at compile time by considering only the bound expressions of the loop instructions.
The shaded regions mark the space described by the loop bounds and the dots show the intersection of these regions and the integer lattice, the actual execution space. One-dimensional operations have 0 as second coordinate. The third coordinate reflects the order of the statements in the source program.

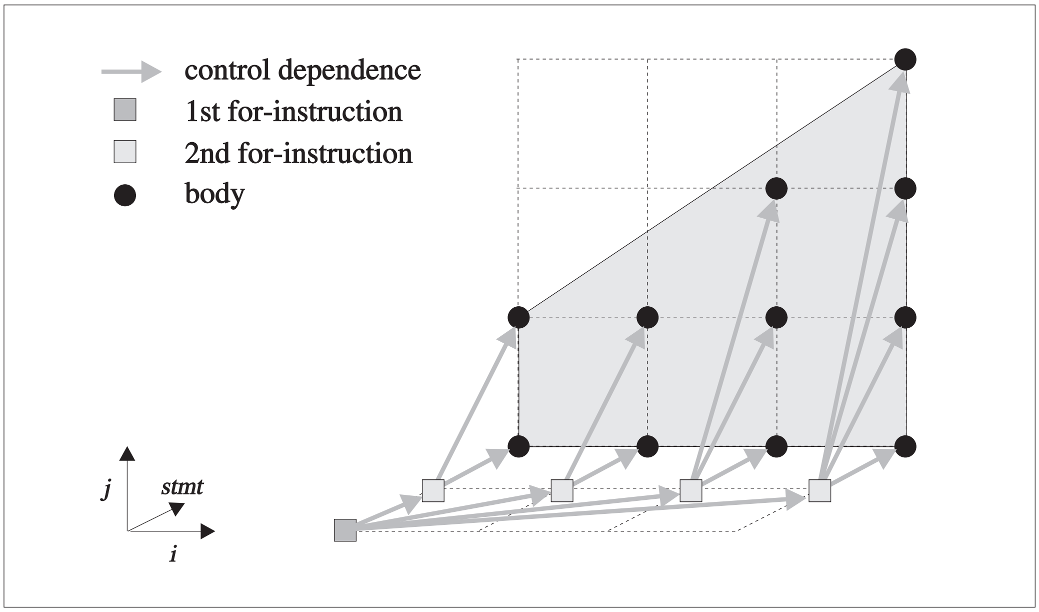

Figure 2.5: The static execution space of the first program in Example 2

The first program yields a polytope as execution space (Figure 2.5):

```
/ * program 1 * /
for i := 0 to 3 do
    for j := 0 to (2/3) * i+1 do
        a[i, j] := 0
    end
end
```

The execution space of the second program is neither convex nor has it straight borders, but it is static, too.

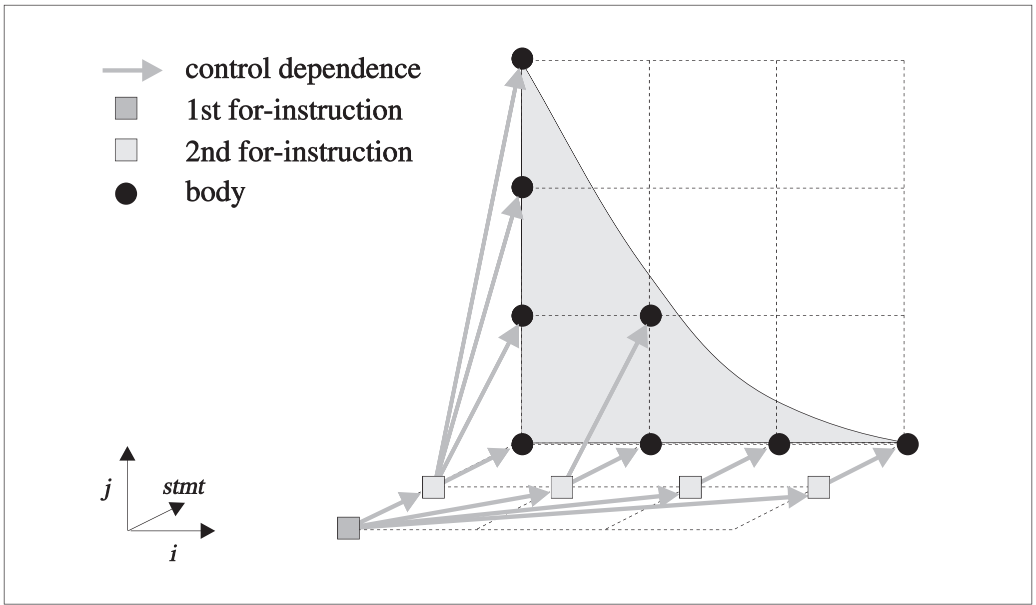

Figure 2.6: The static execution space of the second program in Example 2

To express that static execution spaces do not depend on the body of their loop nests, we can substitute this body by a *skip* statement (the empty statement).

```
/ * program 2 * /
for i := 0 to 3 do
    for j := 0 to (1/3) * (i − 3)^2 do
        skip
    end
end
```

The difference in the shape is not significant for an execution space to be static, but — as we will see in the following sections — it is very significant for target code generation.
Now let us look at dynamic execution spaces. These are characterized by dependences 'pointing' to the bound expressions (for loops) or to the while condition (while loops).

Definition 32 (Dynamic Execution Space).
Let s and s' be two statements in program $\mathcal{P}$. The dimensionality of s is denoted as d_s. Let further bd_ass_r be a bound assignment of a loop surrounding s at level r.
We call the execution space $\mathcal{X}_s$ of s *dynamic* if for some s', where s' is not part of a loop instruction

$$(\exists\, r \,:\, 1 \leq r \leq d_s \,:\, s' \,\delta^d\, bd\_ass_r)$$

In other words: if an execution space is *not static* then it is *dynamic*.

The dependences cause the execution space to change in the course of time. Thus, such execution spaces cannot be enumerated in parallel. Moreover, the upper bound of the time dimension is generally not predictable, and therefore we will obtain while loops for the time dimension in the target loop nest.

Example 3 (Dynamic Execution Spaces).
Dynamic execution spaces can be caused by for loops as well as by while loops.
In program 1 the while loop causes the first dimension to be the "dynamic dimension". A sample execution space is shown in Figure 2.7. Here the array len has the following entries: $len = (1, 2, 0, 1, 6, 4, 2)$
This is a very simple example, but it shows that the extent of the first dimension determined by the while loop depends on the execution of its body and of the while condition of the previous iteration. Therefore the execution space cannot be enumerated in parallel, but has an inherent time component that has to be considered.

```
/* program 1 */
i := 0
for wi := 0 while (len[i] ≤ 5)
    for j := 0 to len[i] do
        a[j] := 0
    end
    i := i + 1
end
```

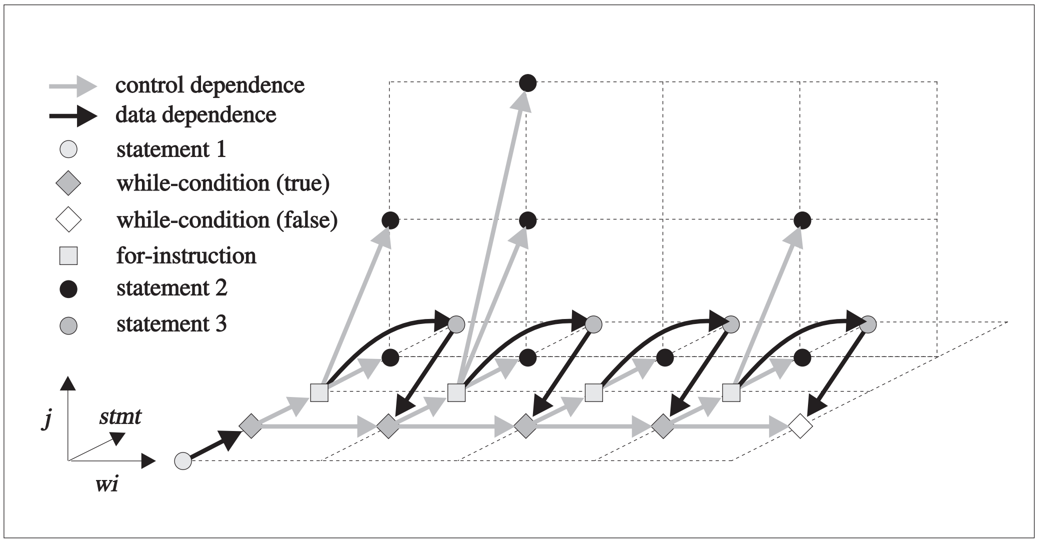

Figure 2.7: The dynamic execution space of the first program in Example 3

We can see that this sequentiality does not affect the second dimension. For every instance of the for loop, all its iterations can be executed in parallel.

As mentioned above, for loops can also cause dynamic execution spaces. This is shown in program 2:
$f(i,j)$ may be any piece of code that computes a value for the array $a[i,j]$.

```
/* program 2 */
a[0,0] := 1
for i := 0 to 3 do
    for j := 0 to a[i,0] do
        a[i+1,j] := f(i,j)
    end
end
```

Again we have a sample execution space, depicted in Figure 2.8. In this example $f(i,0)$ yields the values 1, 2 and 0. These are the only relevant ones for the upper bounds of the second for loop.
As in program 1, every single instance of the second for loop can be executed in parallel. Because the body of instance i of the inner loop affects the bounds of instance $i+1$ of this same loop, the different instances of the inner loop

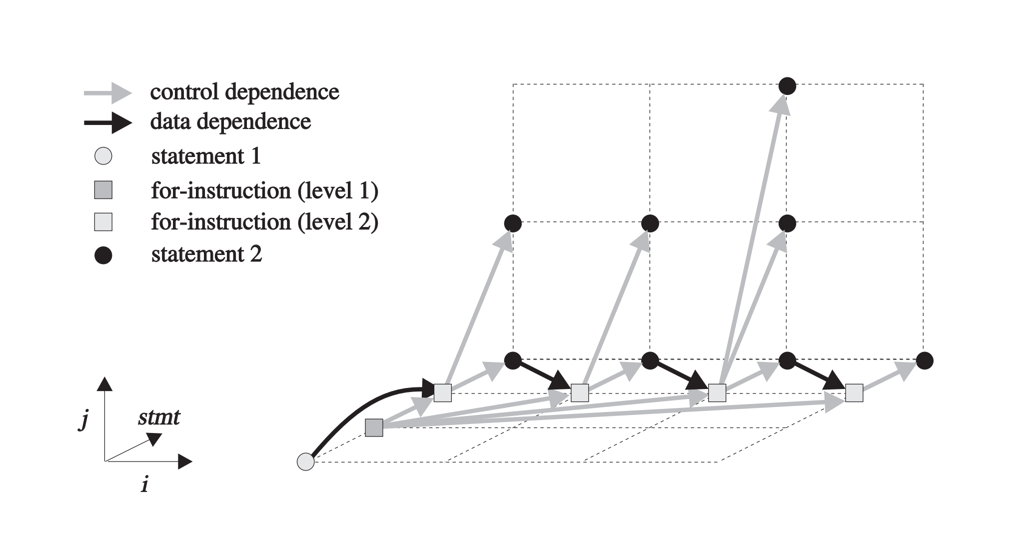

Figure 2.8: The dynamic execution space of the second program in Example 3

have to be carried out one after the other. This means that the outer loop has to be executed sequentially.
Note that, although the extent of the first dimension is known at compile time, it must be enumerated sequentially. Although the upper bound of the second dimension is determined at run time, each instance of the second loop can be executed in one time step.

The explanations in this section make clear that the handling of dynamic execution spaces causes much more overhead than that of static execution spaces, as we have to determine their bounds at run time.

2.3 State of the Art

The present state of the art is that only sequential source programs with loop nests containing only so-called affine for loops can be parallelized using the polytope model. We will briefly describe this kind of for loops and the implications for the source and target execution spaces. For a closer look, we point to [18, 19, 25]. The handling of special cases and some extensions are also described there.
Bound expressions must be affine expressions in structure parameters and indices of enclosing loops (if there are any), i.e., the bounds of a for loop at any level r have the form:

$$\begin{array}{rcl} lb_r & := & \underline{c}_{r,1}i_1+\underline{c}_{r,2}i_2+\cdots+\underline{c}_{r,r-1}i_{r-1}+\underline{c}_{r,r} \\ ub_r & := & \overline{c}_{r,1}i_1+\overline{c}_{r,2}i_2+\cdots+\overline{c}_{r,r-1}i_{r-1}+\overline{c}_{r,r} \end{array}$$

where $\underline{c}_{r,k}, \overline{c}_{r,k} \in \mathbb{Z}$ are constants, $(i_1, \ldots, i_{r-1}) \in \mathcal{X}_{bd\_ass_r}$.
Note that $\underline{c}_{r,r}$ and $\overline{c}_{r,r}$ can be composed of several numerical constants and structure parameters. We view them just as one constant part in the bound expressions.

As constants cannot cause any data dependences (see the remark following Definition 19) and loop indices may only be read in the bodies of the respective loops, the execution spaces caused by affine for loops are always static.
This means that in Figure 2.2 on Page 23 the data dependences pointing to and away from the shaded box of the for instruction do not really exist.
We can determine in a parameterized way at compile time how many iterations the loop will carry out and what values the index will take. Thus, we can also predict the boolean values of the condition in the for instruction at compile time. These are incorporated in the ranges of the target loops, i.e., the target loops will only enumerate the points whose respective conditions in the source loop nest had yielded *tt*.

The shape of an execution space caused by affine for loops is a polytope. Together with an affine transformation function the target execution spaces also result in a polytope. This property guarantees the existence of a precise scan of the target execution space without unnecessary run time overhead.

2.4 Loop Classification

Scanning consists of two tasks: finding bounds for the target loops and making sure that no operations are executed which are not executed in the source program.
The classification proposed in [12] deals only with the second task for for loops; the target loops are considered to be given and we ask what the source execution space must look like to be sure not to enumerate too many target points.
The refinement of while loops into the two classes is not guided by the problem of scanning the target execution space exactly, but is done with respect to existing related work, such as [22, 24, 28].

Remark. The titles of the following sections give the numbering of the classes introduced in [12].

2.4.1 Affine for Loops (Class 4 Loops)

This class is equal to the class described in the last section. In difference to Class 3 (see Section 2.4.2) we always know how to compute the bound expressions for the target loop nest.

2.4.2 Convex Execution Spaces (Class 3 Loops)

The for loops contained in Class 3 ensure convex target execution spaces, i.e., they can be proved at compile time to cause only convex execution spaces. This "proved at compile time" induces that only static convex execution spaces are considered here. The class of convex loops forms a superset of Class 4, as the latter always yields (polytopes as a special case of) convex static execution spaces.
The definition of Class 3 makes use of the fact that an affine function maps a convex set onto a convex image: if the bounds of the image are given as a function of target loop indices and structure parameters, it can be scanned exactly.

Example 4 (Convex Execution Spaces).
The following program has the convex static execution space (of loops in Class 4 and Class 3) depicted in Figure 2.9. The upper bound of the inner loop prevents the loop nest from belonging entirely to Class 4.

```
for i := 0 to 4 do
   for j := 0 to sqrt(4 * i) do
      body
   end
end
```

In Class 3 we free ourselves from the question: "How can we find the bound expressions ?" We simply say: "If someone gives us the bound expressions, we know the target execution space to be scannable (precisely) !"

Remark. We could also imagine a more practical view of Classes 3 and 4 by defining one 'big' class of convex loops which contains the class of affine loops and the subset of Class 3 for which we actually have the mathematical tools to find functions that describe the shape of their target execution spaces under any affine transformation. Thus, a for loop with a bound that causes a convex static execution space would not be part of the new convex class if we were not able to find a function as bound for the target loops that yields the respective border of the target execution space.

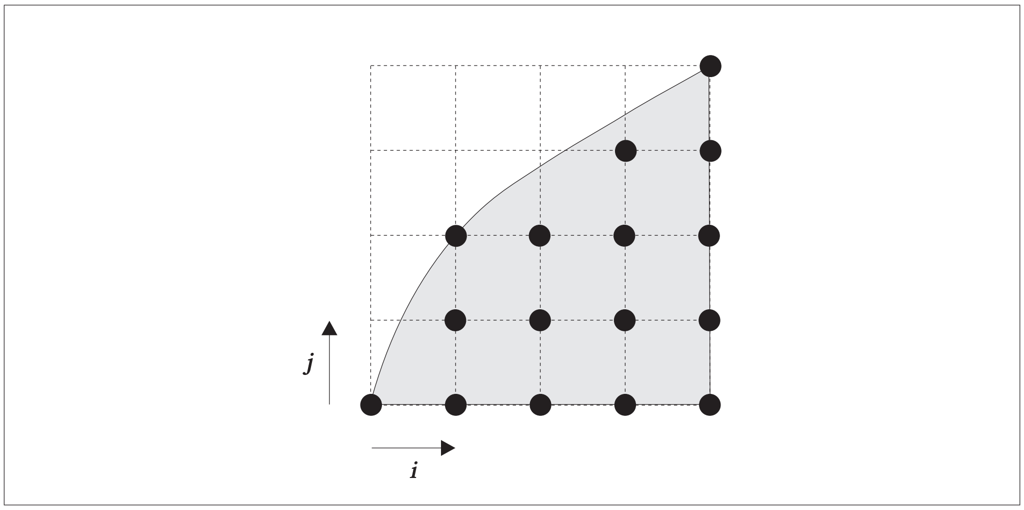

Figure 2.9: The convex static execution space of the program in Example 4

2.4.3 Arbitrary for Loops (Class 2 Loops)

Class 2 contains all other types of for loops. In general, they yield unscannable dynamic execution spaces that must be estimated at run time. We will show this estimation in the next chapter.

Example 5 (Class 2 Loops).
We already saw examples for programs with Class 2 Loops. In Example 3 on Page 30 the inner for loop of program 2 belongs to Class 2. As it is dynamic, it cannot be proved at compile time to yield only convex execution spaces.
The inner loop of program 2 in Example 2 on page 28 also belongs to Class 2. It is static, but its upper bound yields a concave execution space.

2.4.4 Static while Loops (Class 1 Loops)

The number of iterations of while loops in Class 1 is fixed before the first operation of the body is executed, but it is not given explicitly. It is computed iteratively instead.
Different approaches to parallelization (e.g., [28]) rewrite the source loop to first compute the number of iterations sequentially and than carry out the remainder of the body in parallel.

Example 6 (Class 1 Loops).
See Example 3, program 1. The while loop is static, because the values

of the variables in the condition are not changed in the body. Thus, the number of iterations of the while loop does not change during its execution, but it is not given explicitly and is not computable in O(1). We cannot determine by looking only at the loop instruction whether the array $len[i]$ is updated somewhere else in the program. So the number of iterations cannot be computed at compile time, either.
Note that the 'static' in the term "static while loops" has nothing to do with the 'static' in "static execution space". A static while loop always causes a dynamic execution space.

2.4.5 Dynamic while Loops (Class 0 Loops)

In contrast to Class 1, the number of iterations of a loop in Class 0 is affected by its body, i.e., a while loop of Class 0 changes its number of iterations during its execution. In the setting of the alternative approach mentioned for Class 1 this means that the subsequent parallel forall loop could be empty.
In our setting we do not distinguish Class 1 and Class 0. We believe that our result of parallelizing nests of Class 1 loops without further inspection is not worse than the alternative, specialized approaches and yields a better load balance of the processors.

Note that with additional semantic knowledge our classification collapses, as every loop may move from a more general class to a more specialized one. Looking at Class 2, we could also imagine to have convex execution spaces that are *dynamic*. However, to prove the convexity at compile time we would need additional semantic information about the program and the data. Some approaches examine such properties and try to parallelize the easier loop found this way [22, 28], but this is exactly what we do not want to consider in this classification.
Our approach is more general. We base our analysis on a mathematical foundation rather than finding special cases ad hoc that can be parallelized. However, our method does not prevent the usage of such methods. If we knew more about our source program we possibly could use methods for easier loop classes and exploit more parallelism this way.

2.5 Consideration of the Transformation Function

The classes of the previous section were defined without considering transformations. We involve these at the next level of analysis.

In [10] a property of space-time mappings is defined which guarantees scannable target execution spaces. This property is not of interest for Classes 3 and 4, because the shape of their execution spaces ensures scannable target execution spaces anyway.
Scannable transformations ensure the following two aspects of scanning:

- The execution space is only changed in a way that the target execution space has "*no holes inside*".
- The target loops can describe the bounds of the target execution space precisely.

These advantages of scannability hold for both static and dynamic execution spaces. We refine the static and dynamic parts of Class 2 into one that is determined by scannable transformations and one that contains the rest.
Griebl [9] proves that for every asynchronous target program a scannable transformation can be found. This is nearly never the case for synchronous target programs.
Scannability implies that no target loop at level r may depend on a source loop at a level greater then r. Thus, if we have a one-dimensional schedule for a synchronous target program, only the outermost source loop may be a dynamic loop. Or the other way round: a valid scannable transformation for a synchronous target program is in general only possible if the dimension of the schedule is at least as large as the level of the innermost dynamic loop in the source loop nest.
Typically this will prevent much of the parallelism allowed by the source program, and therefore a non-scannable transformation will probably be chosen.

Remark. If there are if statements in the source program there is in general no scannable transformation. if statements mean that the source execution space "has holes" that are arbitrarily spread across the index space[6]. The target execution space has these holes, too, and we have to test for them at run time.

To conclude this section: scannability of transformations compensates for non-convexity of the execution space and makes it 'easy' to find target loop bounds. Essentially the original loop bounds only have to be transformed by the space-time mapping.
The opposite side of the coin is that (at least for synchronous target programs) we cannot always find preferable scannable transformations.

[6] This is different to the 'holes' produced by strides different to 1 or -1 that are spread across the index space in a regular pattern.

2.6 Additional Consideration of the Target Execution Space

The target execution spaces related to affine and convex loops and to transformations that are scannable can be scanned precisely without actually looking at their shape. Now the question arises, what we could tell about the scannability of static target execution spaces if we knew their shape and extent in addition to the source execution space and the transformation.

2.6.1 Scannable Target Execution Space

We can find target execution spaces that are scannable, though their space-time mappings are not scannable and their source execution spaces are not convex, either.

Example 7 (Consideration of Target Execution Spaces).
Figure 2.10 shows on the left side the non-convex static execution space of the following sample program:

```
for i := 0 to 5 do
    for j := 0 to ((i − 5)^2)/5 do
        skip
    end
end
```

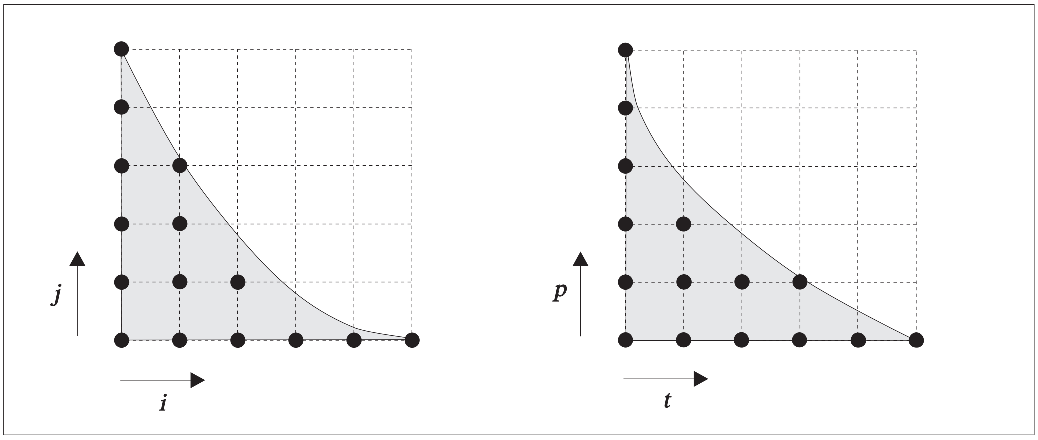

Figure 2.10: The execution spaces of the program in Example 7

The right side shows the target execution space determined by the space-time mapping $T = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$.
Although T is not scannable and the execution space is not convex, the scanning of the target execution space causes no problems — the target execution space is scannable.
To get the bounds of the target loops we essentially have to compute the inverse function of the upper bound of the second loop. Therefore we have to solve the following inequalities for p:

$$t \leq \frac{(p-5)^2}{5}$$

For the given range of p: $0 \leq p \leq 5$ we get:

$$p \leq 5 - \sqrt{5t}$$

We can use the following target loop nest to enumerate the target execution space:

```
/* target program */
forall t := 0 to 5 do
   forall p := 0 to 5 − sqrt(5 * t) do
      skip
   end
end
```

2.6.2 Minimal but not Exactly Scannable Target Execution Space

Due to non-convex execution spaces, there are cases where the smallest possible scanned target space is bigger than the actual target execution space. This is possible because a loop cannot skip some iterations and then go on to continue the execution of its body.

Example 8 (Minimal but not exactly scannable Target Execution Space).
This example is a continuation of Example 7. We have the same program with the same execution space, but this time we consider a different transformation function, $T = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$, with its corresponding target execution space (depicted in Figure 2.11). Again T is not scannable and now the target execution space is not scannable, either.

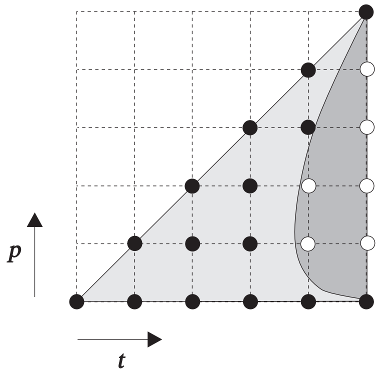

Figure 2.11: The target execution space for the transformation in Example 8

The light grey region represents the transformed target execution space. The darker grey region contains the additional points that have to be scanned, but are not in the target execution space. Filled dots represent iterations which have to be executed, white dots mark iterations that are scanned but must not be executed.
It is not possible to find a loop nest that scans a smaller region without loosing points belonging to the target execution space.

The difference to the previous class is that, although we know the upper border of the second dimension *and* its image, we cannot find a loop nest which scans a smaller region without omitting points of the target execution space. Thus, we have to take care of points whose inverse images are not in the execution space.

Remark. The same holds for execution spaces that are determined by if statements.

2.6.3 Approximatable Target Execution Space

For the class described in the previous subsection we had to scan too many points because *loop nests* were not powerful enough to describe the target execution space exactly. In addition to this, the *class of approximatable target execution space* deals with the lack of mathematical procedures or tools for describing the bounds of the target execution space as (closed) functions in target loop indices and constants.

For our classification it does not matter whether we have no tool or whether it is theoretically impossible to find a function that describes the exact target execution space. In both cases the consequence is that we have to approximate the target execution space by a superset which liberates us from the actual shape and which we can describe. This approximation can be done at compile time for static execution spaces.
For instance, a polytope may serve as an approximating superset, because for any polytope we have a method for enumerating its image under an affine space-time mapping.
We use the following property of affine functions (omitting the proof) that ensures that we do not skip points of the target execution space if we scan the image of a superset of the execution space.

Lemma 33. *Let $\mathcal{X} \subseteq \mathcal{XS} \subseteq \mathcal{I}$ and T an affine function $T : \mathcal{I} \longrightarrow \mathcal{TI}$ with $T(\mathcal{X}) = \mathcal{TX}$ and $T(\mathcal{XS}) = \mathcal{TS}$. Then: $\mathcal{TX} \subseteq \mathcal{TS} \subseteq \mathcal{TI}$*

The following example shows the characteristic features of approximatable target execution spaces.

Example 9 (Approximatable Target Execution Spaces).
Let the following program be given and assume that we have no tool to find the bounds of its target execution space under any transformation. The left part of Figure 2.12 shows the execution space, the right part shows the target execution space.

```
for i := 0 to 12 do
    for j := 0 to (i/2) + 2 * sin(i) do
        skip
    end
end
```

We can approximate the execution space (light grey) by a polytope (union of the light grey and dark grey region) yielded by the following loop nest:

```
/ * approximation * /
for i := 0 to 12 do
    for j := 0 to (i/2) + (5/2) do
        if(j <= (i/2) + 2 * sin(i))then
            skip
        end
    end
end
```

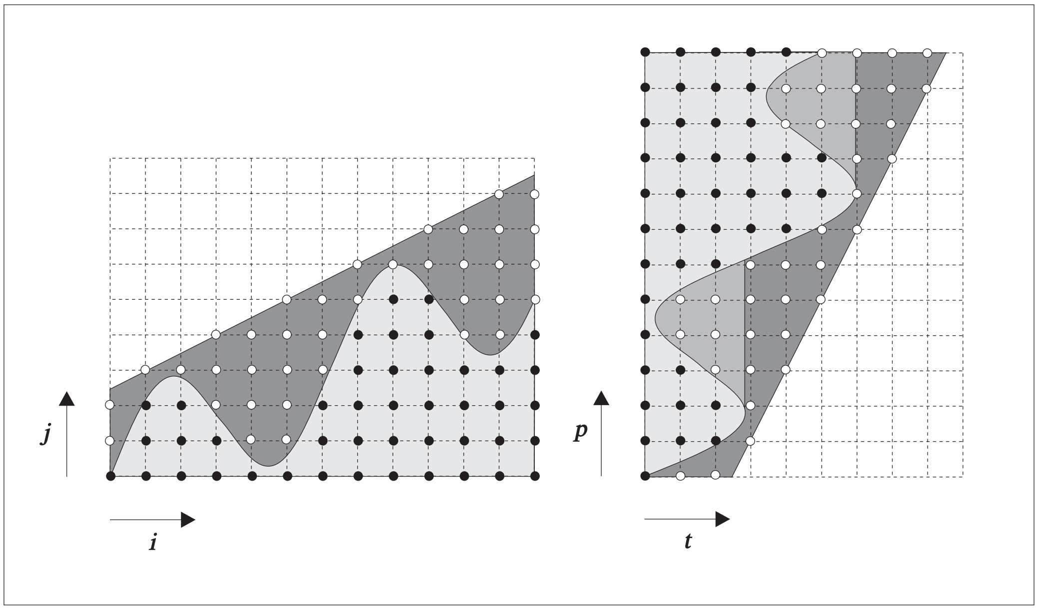

Figure 2.12: Execution- and target execution space of the program

Since we enumerate too much, we have to take care of the points that are scanned but are not in the execution space (white points). This is done on the fly by the if clause.
The altered program can now be transformed with the methods for polytopic execution spaces.
The right part of Figure 2.12 shows the target execution space (light grey), the minimal number of points which must be scanned additionally (medium grey). The dark grey region contains the additional points caused by the approximation of the execution space. Again, filled dots must be executed and white ones must not.
The medium grey region is caused for the same reasons as discussed in Subsection 2.6.2, i.e., because of the non-convex execution space.

Remark. Note that the approximation by an affine upper bound is only an example. In general, any function that yields a superset of the execution space may serve as an approximation. However, it should be a function that ensures an approximation which can be scanned precisely.
In contrast to the class described in Section 2.6.2, we have to go to the trouble of finding an appropriate approximation.

Without further semantic information about the program we cannot take the shape and extent of dynamic target execution spaces into account for our analysis, as they are only known at run time (see the dark grey box in Figure 2.1).

2.7 Implementation in LooPo

The last row in Figure 2.1 shows the classes we distinguish for our implementation in the automatic parallelizer LooPo.
Current methods can already deal with the class of (extended) affine loops that yield polytopes and quasi-convex polytopes as execution spaces.
We treat all other for loops as if they were dynamic for loops and all other while loops as if they were dynamic while loops. This worst-case view ensures that we can put up with all kinds of sequential loops, but by nature we cannot treat all cases in the best possible way. These optimizations are up to future work and further studies.

Chapter 3

Dynamic Approximation of Execution Spaces

As we mentioned in the last chapter, we distinguish between affine for loops with static execution spaces, other for loops (static or dynamic) and while loops (always dynamic). The present stage of development of the polytope model is already suited for parallelizing nests of affine for loops. In the current chapter we extend the theory to dynamic execution spaces. For the integration of while loops we base our work on Griebl [9] and adapt it for imperfectly nested loops.

Dynamic execution spaces can only be calculated at run time. In general, this computation cannot be precise (see Section 2.6) but must yield a superset, the scanned target space.
If we scan a superset of the target execution space, we have to take care not to execute points whose inverse image does not belong to the source execution space. We explain this execution analysis in Section 3.2, but first of all we present a simple example that demonstrates the problems we are facing.

3.1 Example

Consider a program with one surrounding affine for loop and a non-affine for loop that determines the second dimension of the body statement s. Assume that the transformation we use, preserves the dependences of the loop nest.

```
for i := 0 to 5 do
   for j := 0 to a[i] do
      s
   end
end
```

The inequalities describing the execution space of the body s would be:

$$\begin{array}{rcrcrcl} i & & & & & \geq & 0 \\ -i & & & + & 5 & \geq & 0 \\ & & j & & & \geq & 0 \\ & - & j & + & a[i] & \geq & 0 \quad (\forall\, i : 0 \leq i \leq 5) \end{array}$$

As we might not know the values of $a[i]$ at compile time[7], we must omit the last inequality and approximate the execution space by a polyhedron whose second dimension is unbounded. Later, when we are generating the target code, we have to determine an upper bound for the target loops depending on j.

Let $T_s = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$ be the affine synchronous transformation. Its inverse is $T_s^{-1} = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}$.

Thus, the following inequalities define the target space, where t is the index for the time and p the index for the space dimension:

$$\begin{array}{rcrcrcl} t & - & p & & & \geq & 0 \\ -t & + & p & + & 5 & \geq & 0 \\ & & p & & & \geq & 0 \end{array}$$

To get a synchronous target program, we have to express p in terms of t:

$$\begin{array}{rclcl} t & \geq & 0 & & \\ p & \geq & 0 & & \\ p & \geq & t & - & 5 \\ p & \leq & t & & \end{array}$$

This leads to the target loop nest

[7] If the loop of i were also non-affine or a while loop, we would not even know an upper bound for i at compile time.

```
/* synchronous target program */
for t := 0 to ? do
   forall p := max(0, t − 5) to t do
      s'
   end
end
```

where s' is derived from s just by expressing i and j in terms of t and p. The unknown upper bound $a[i]$ of j in the source inequalities implies that we do not know an upper bound for the time t now. It would have to be

$$\max\{\tau_s(i,j) \mid 0 \leq i \leq 5 \quad \wedge \quad 0 \leq j \leq a[i]\}$$

We do not know the values of $a[i]$ and so this upper bound can only be computed at run time step by step as the values become known. This "step by step" can be formulated as a while loop for the time.

Let us take a look at the asynchronous target program now. To achieve this, we have to express t in terms of p and we get the following description of the target space:

$$\begin{array}{rcl} p & \geq & 0 \\ t & \geq & 0 \\ t & \geq & p \\ t & \leq & p + 5 \end{array}$$

We transform the inequalities to an asynchronous target program and get:

```
/* asynchronous target program */
forall p := 0 to ? do
   for t := max(0, p) to p + 5 do
      s'
   end
end
```

Again we do not know the upper bound of the outermost target loop, but if we changed this loop into a while loop, we would have a completely sequential target program, as there are no parallel while loops [9][8]. We cannot compute

[8] We do not consider partitioning in this example, but describe its implications in Section 3.5.

the upper bound of the outermost loop iteratively at run time, because the semantics of a for loop (and a forall loop) prescribes that the bounds are fixed once and for all before the body is executed; they never change during execution, even if the body changes the value of a variable occurring in one of the bound expressions.
We notice that now there is a *problem-inherent* difference between the time and space dimensions; formerly, when we only had affine for loops, there was only a technical difference for code generation, namely that space dimensions were enumerated by parallel forall loops whereas time dimensions were enumerated by sequential for loops to preserve the dependences in the source program.

If the second loop were a while loop, e.g,

$$FOR\ j := 0 \ \textsf{while}\ (a[i] < 10)\ \textsf{do}\ body\ \textsf{end},$$

we also would not know the upper bound and we would have to drop the last inequality. This would result in the same target loops as for the non-affine for loop. However, we shall see that the calculation of the appropriate upper bound is more complicated.

Together with the implications of (un)scannability (see the previous chapter) we make three observations:

1. In general (without partitioning) we can derive *only synchronous target programs* that may have while loops for enumerating the time.

2. We have to take care of the termination of the target loops and call this task *termination detection.*

3. As target execution spaces are *unscannable* in general, we have to scan a superset of the actual target execution space. Thus, we have to decide which target points must be executed and which must not. We call this decision *execution determination.*

3.2 Execution Determination

This chapter examines how execution determination can be realized. We choose to describe this before termination detection, because we implement termination detection as side effect of the execution determination (Section 3.3).
Before we introduce a general predicate *executed* for every statement in a source program we include if statements in our considerations.

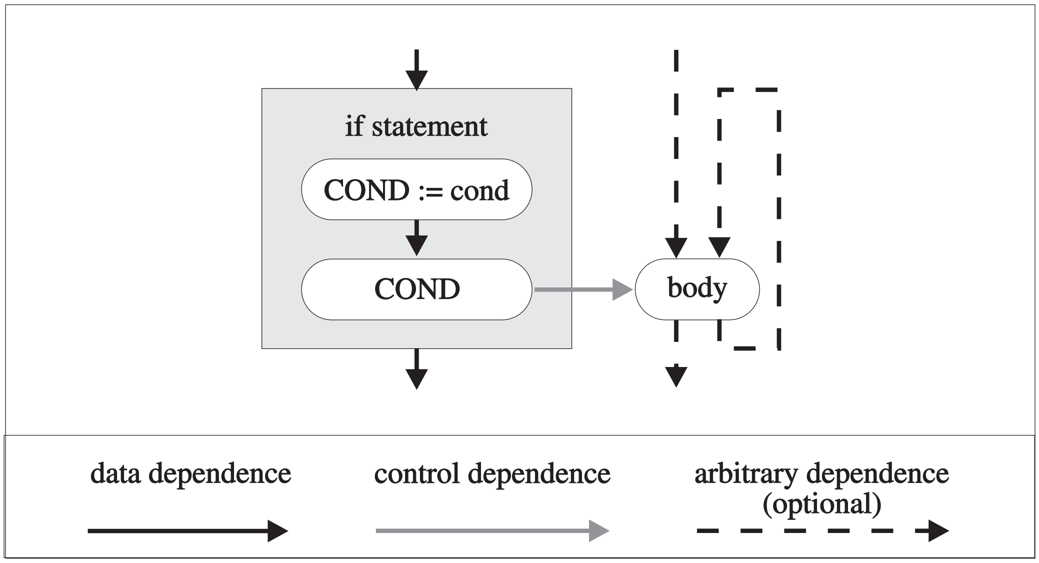

Figure 3.1: Dependences related to an if statement

3.2.1 Execution Condition of an if Statement

In Chapter 2 we define the execution space of a statement without considering if statements, because we do not want to include if statements in the classification. However, the execution space of a statement is determined not only by the surrounding loops but also by surrounding if statements.
Figure 3.1 shows an if statement with the possible dependences.
The substatement $COND := cond$ expresses that the condition is evaluated once and the truth value remains the same for every statement in the body of the if statement ($COND$ is only *read* from now on) even if some statement changes a variable occurring in *cond*.
Data dependences can be caused by the variables in the condition. The essential purpose of an if statement, however, is the implementation of a control dependence to the statements in its body. We can also put this the other way round: a control dependence can be implemented by an if statement.
Like for loops we can also find execution conditions for if statements.

Definition 34 (Execution Condition of an if Statement).
The execution condition of an if statement is the if condition itself. Denoted: ex_condif_k, where in this case k is a unique number for each if statement.

An operation may only be executed if all surrounding execution conditions evaluate to tt.

In Chapter 4 we describe the implementation of if statements in LooPo.

3.2.2 General Predicate *executed* for each Statement

Until now we have considered the execution conditions caused by loops and if statements only dimension by dimension. In this section we combine these separate conditions to one condition for each statement that determines whether an operation of this statement is to be executed or not.

Definition 35 (Predicate *executed*).
Let $s \in \mathcal{S}_{\mathcal{P}}$ be a statement in source program $\mathcal{P}$ and d_s the dimensionality of s. Let M be the number of surrounding if statements of s. Then the predicate $executed_s : \mathcal{I}_s \rightarrow \{ff, tt\}$ determines whether an operation of statement s is executed at a given iteration. We define:

$$\begin{array}{rclcccccc} executed_s(i_1, \ldots, i_{d_s}) & := & ex\_condl_1 & \wedge & \ldots & \wedge & ex\_condl_{d_s} & \wedge \\ & & ex\_condif_1 & \wedge & \ldots & \wedge & ex\_condif_M & \end{array}$$

where $ex\_condl_k := ((ex\_condf_k \wedge is\_for(i_k)) \quad \vee \quad (ex\_condw_k \wedge is\_whl(i_k)))$, $1 \leq i_k \leq d_s$.

Remark. The sequence of the various execution conditions does not matter for the theoretical definition. However, for the implementation we have to interleave the execution conditions of if statements with those of the loops according to their original nesting order in the source program.

All the definitions concerning execution conditions are phrased only in terms of the source side. In the target program, however, the source indices are no longer visible.
Since we consider only bijective affine transformation functions in this thesis, every source index can be uniquely recomputed from an affine combination of target indices and constants.
Let T_s be the transformation function of a statement s and T_s^{-1} its inverse. Then:

$$i_k = \Pi_k(T_s^{-1}(i'_1, \ldots, i'_{d'_s}))$$

where $i_k \in \mathcal{E}_s$ are source indices, $1 \leq k \leq d_s$, and Π_k is the projection onto the k-th dimension, i.e., the k-th row of the matrix T_s^{-1}.
If we replace the source indices in all *ex_condl*, *ex_condif* and $executed_s$ by these expressions we can check whether a target point must be executed or not.

We have presented a way to ensure that only target operations of a statement are executed that have an inverse image in the respective source execution space. In the next section we take care of the bounds of the target loops and show their termination and that they enumerate a large enough target space.

3.3 Termination Detection

Example 3.1 shows that we have to find 'suited' bounds for the target loops. Obviously this is not possible with the original loop instructions, so we propose a rewriting of the source program to get index spaces we can handle. Unfortunately these new index spaces are not equal to the old ones, but bigger in general. Thus, we have to take care of the additional points as described in Section 3.2.

3.3.1 Scanning Execution Spaces of Dynamic for Loops

In this section we describe how we transform dynamic for loops to yield index spaces we can handle.
Figure 3.2 depicts the changed for instruction for a non-affine for loop[9].
Essentially two things have changed:

1. We have got a new substatement in the part of the for instruction which computes the new upper bound of the loop.

2. We guard the original body with the execution condition of the original loop.

The new substatement $max\_ub_r := \max(max\_ub_r, UB_r)$ is responsible for the 'inexact' iteration space but yields the benefit of getting a monotone function for the upper bound. The upper bound of an instance of the loop is at least as high as the maximum of the upper bounds of all previous instances of this loop. This monotony is one of the reasons why affine bound expressions are so suitable for parallelization. It enables us to compute the minimum and maximum easily: we find the extrema of a monotone function either at the beginning or at the end of the source domain.

The price for a convenient shape of the new index space is firstly the effort that must be taken to prevent the execution of operations that do not belong to the execution space. By guarding the body with the original execution

[9]Note that Figure 3.2 only shows the situation for a positive stride. If ST_r is negative the computation of the maximum has to be replaced by the computation of the minimum.

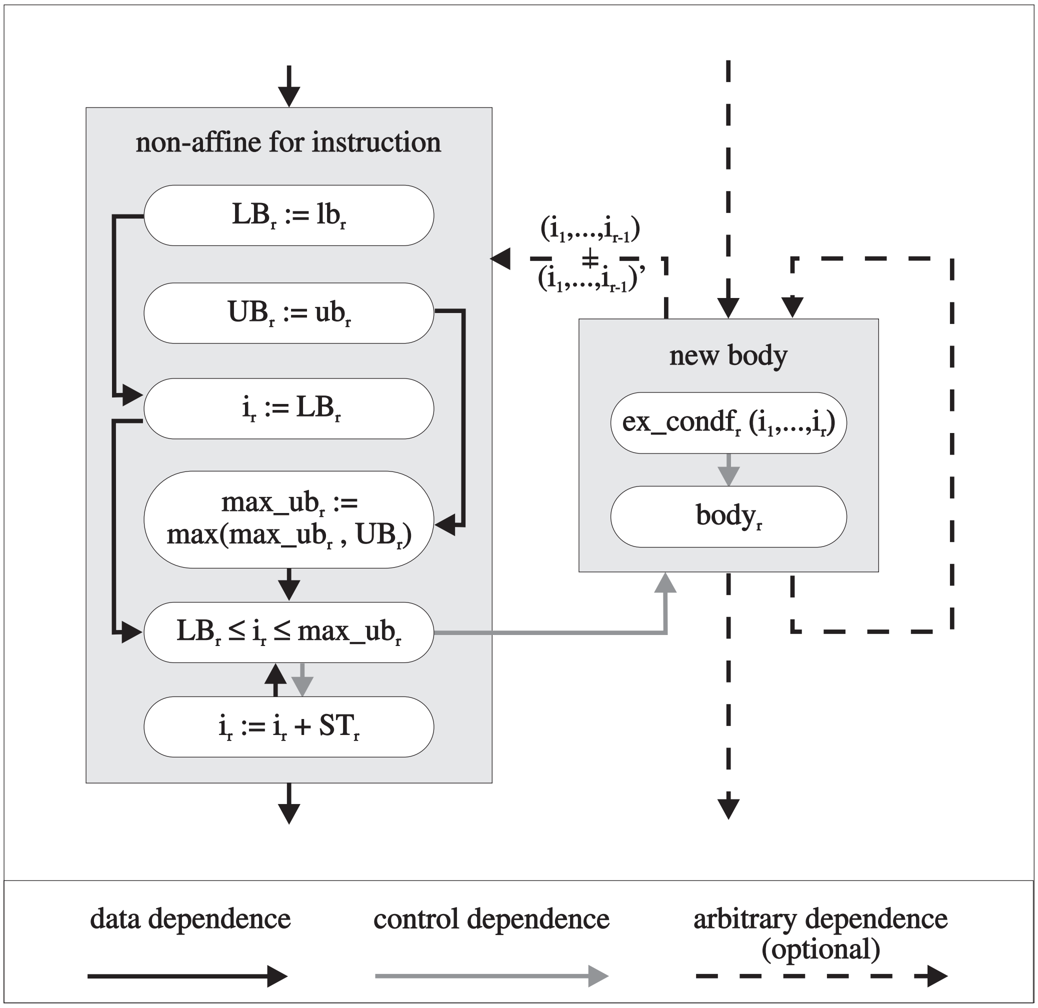

Figure 3.2: Changed non-affine for loop with a new upper bound

condition we ensure that the execution space of the body statements remains the same, although the number of iterations of the loop has changed (see also Section 3.2).

Secondly, the computation of the maximum introduces a new sequentiality that might not have existed in the original source program. Note that instances of a loop at level r can still be executed in parallel, if this has also been possible in the original program. The sequentiality is affecting the loop at level $r-1$.
If we look at Figure 3.2, however, we notice that there is no dependence from an instance of the max_ub_r-statement to its next instance. This is possible,

because we have some degree of freedom here.
In the model, the missing dependence allows all operations of the max_ub_r-statement to be executed in parallel. This approach makes sense, because the order in which the values are compared is not important and we can leave it up to the hardware (or the memory management) to synchronize these memory accesses. The calculation of the maximum of n values is done in one step of time from the view of the model, but its real duration naturally depends on the value of n and refines the time steps. As a consequence, from now on there is no (strong) correlation between the schedule and the running time of a program anymore.
If the target program is intended to run on an asynchronous target machine, we have to ensure that all updates of max_ub_r that are scheduled at the same time are done before the new upper bound is used.

3.3.2 Scanning Execution Spaces of while Loops

While the number of iterations of an instance of a for loop is known before the first operation is executed, the number of iterations of a while loop is calculated during its execution. This is the main difference between non-affine for loops and while loops. The consequences are examined in this section.

In contrast to for loops, the upper bound of a while loop is not given as a value but as a condition. When we are dealing with a while loop, we have to compute an upper bound (a number) that is increasing monotonously over all instances of this loop. Figure 3.3 shows the changes of a while loop in order to achieve this goal.
Similar to non-affine for loops (Section 3.3.1), we need a new variable that expresses the upper bound. For consistency reasons we also call it max_ub_r. It is shared by all instances of one while loop.
In the beginning max_ub_r is set equal to the lower bound. This means that every instance of the while loop evaluates the predicate *executed* at least once and this is equivalent to the semantics of a while loop, which prescribes that the condition is evaluated at least once.
As long as the condition $LB_r \leq i_r \leq max\_ub_r$ holds, the respective instance of the loop is iterating. $i_r \leq max\_ub_r$ means that every instance is running at least as long as the previous instance. If the current instance turns out to execute more iterations than the previous one, max_ub_r is incremented by 1 at every new iteration (see the darker grey box in Figure 3.3). Thus, at any given point in time, max_ub_r stores the maximum number of iterations which an instance of the loop has carried out up to now.

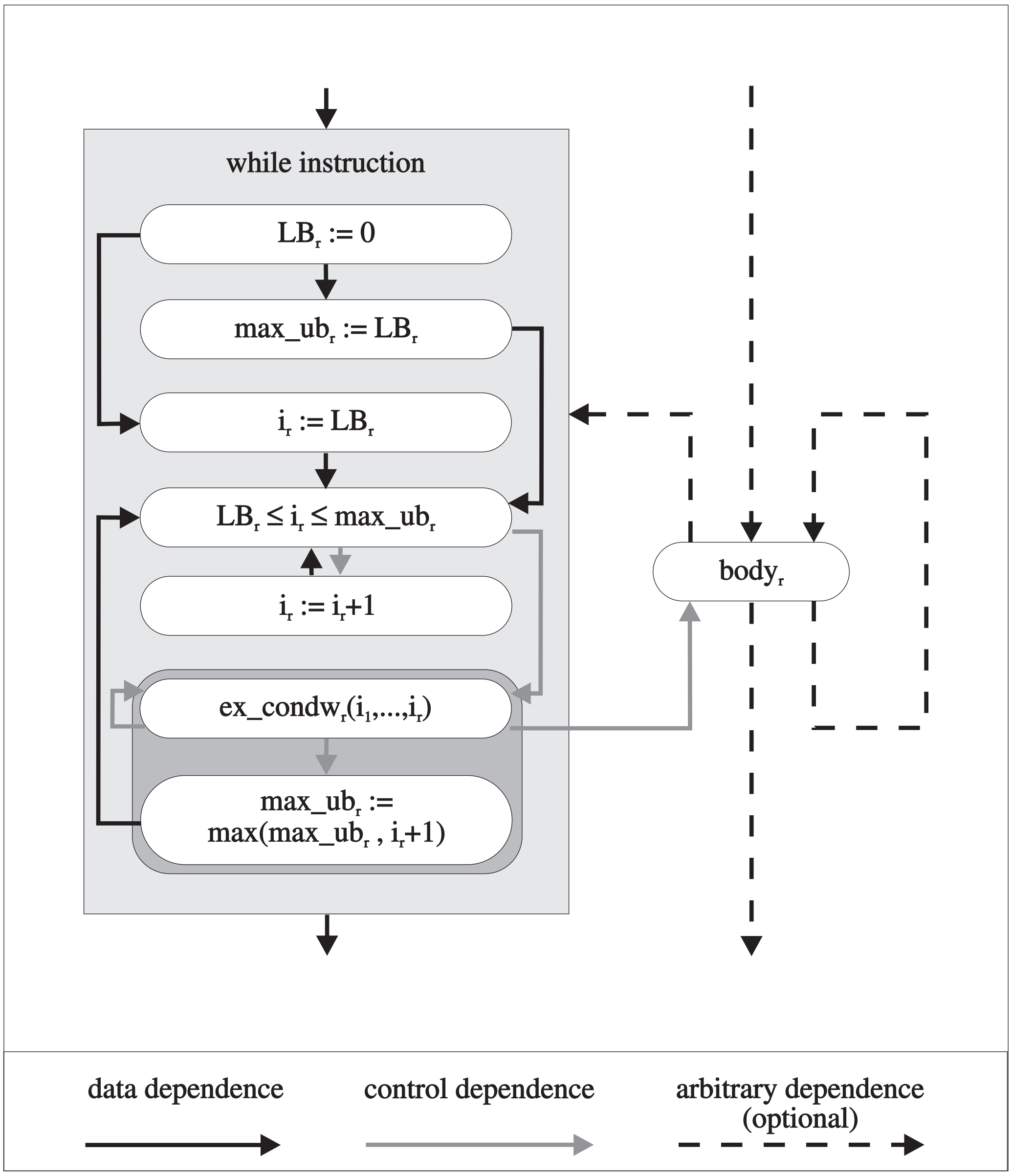

Figure 3.3: Changed while loop with an upper bound

Since every instance of the loops runs at least as long as the previous one, some instances run longer than they would have done originally. This is the reason for the difference between the iteration and the execution space and therefore we have to take care of execution determination expressed by the truth value of *executed*.

3.3.3 Bounds and Termination

We have presented how we want to change the original source program and claim that we can handle the changed program. Before we provide the proofs of correctness and applicability, let us look at the following lemma.

Lemma 36.
*Let s be a statement in the body of a non-affine (*for *or* while*) loop at level r and $max\_ub_r(\tau_s(\bar{x}))$, $\bar{x} = (\bar{i}_1, \ldots, \bar{i}_{d_s})$, the value of max_ub_r at time $\tau_s(\bar{x})$. Then the currently known value of the upper bound of loop r, $UB_r(\bar{i}_1, \ldots, \bar{i}_r)$, is already considered for the calculation of max_ub_r at a time earlier than $\tau_s(\bar{x})$. Further, we can abstract from the inductive computation of max_ub_r (Figures 3.2 and 3.3) by looking at the values which are actually considered. Formally:*

$$(\forall\, x = (i_1, \ldots, i_{d_s}), \bar{x} = (\bar{i}_1, \ldots, \bar{i}_{d_s}) \,:\, x, \bar{x} \in \mathcal{I}_s \wedge \tau_s(x) \preceq \tau_s(\bar{x}) \,:\, max\_ub_r(\tau_s(\bar{x})) = \max\{UB_r(i_1, \ldots, i_r)\})$$

Remark. For a for loop at level r $UB_r(i_1, \ldots, i_r) = UB_r(i_1, \ldots, i'_r) = UB_r(i_1, \ldots, i_{r-1})$, where $i_r, i'_r \in \mathbb{Z}$. while loops need the last coordinate i_r, as the upper bound changes during execution. To be uniform we choose to use it for for loops, too.

Proof (Sketch).
The (transitive) dependences from the max_ub_r-statement to every body statement ensure that the operation $\langle max\_ub_r, (\bar{i}_1, \ldots, \bar{i}_r)\rangle$ is executed at a time earlier than $\tau_s(\bar{x})$, i.e., that the $UB_r(\bar{i}_1, \ldots, \bar{i}_r)$ are considered "early enough". As for the result of a max-instruction, the order in which the values are considered does not matter; at a given time $\tau_s(\bar{x})$ the variable max_ub_r contains the specified value.

In the following part of the thesis we explain how and prove (partly informally) that we can find expressions for the bounds of the target loops. For simplicity, we assume that the lower bounds of non-affine loops are smaller than the respective upper bounds, i.e., that the strides are positive. This requirement is not a limitation of our method, but it eliminates some case distinctions.

First of all we need some requirements to be satisfied by the source program, the scheduler and the allocator:

- *Requirement 1*: The lower bounds of non-affine loops have to be affine expressions in the indices of surrounding loops and parameters (we shall see later that this is *not a* severe limitation).
- *Requirement 2*: Let i_r and i'_r be two values of the loop index of a non-affine loop at level r and $body_r$ its body. Then we require:

  $$\begin{array}{c} (\forall\ s\ :\ s \in body_r\ : \\ i_r \leq i'_r \quad \Leftrightarrow \quad \tau_s(i_1,\ldots,i_r,\ldots,i_{d_s}) \preceq \tau_s(i_1,\ldots,i'_r,\ldots,i_{d_s})) \end{array}$$

  This means that statements in the body of a non-affine loop must not be enumerated in opposite order, even if the dependences allow this. For while loops, this requirement is always satisfied because of the while dependence, but since we do not insert a similar dependence into for instructions, Requirement 2 must be imposed on non-affine for loops.
- *Requirement 3*: Upper bounds of non-affine for loops must not be considered as known, i.e., there is no inequality that limits the upper bound of the index of a non-affine for loop.

With these requirements, it is always possible to find lower bounds for the time loops that do not depend on the upper bounds of non-affine loops (which we do not know at compile time).

Definition 37 (Minimum/Maximum of Time for a Statement).
Let $s \in \mathcal{S}_\mathcal{P}$ a statement in program $\mathcal{P}$ and τ_s the affine schedule of s. Then

$$\begin{array}{rcl} min\_t_s & := & \min\{\tau_s(x) \mid x \in \mathcal{S}_s\} \\ max\_t_s & := & \max\{\tau_s(x) \mid x \in \mathcal{S}_s\} \end{array}$$

are the minimal and maximal values for the time at which an operation of s is executed.

Lemma 38 (Lower Bounds for Time Loops).
Let τ_s be the affine schedule of a d_s-dimensional statement s in the body of a loop nest with non-affine loops and let NAL be the set of loop indices that

belong to a non-affine loop. Then the lower bound of the range of time for the operations of s is:

$$\begin{array}{c} min\_t_s = \min\{\tau_s(MB_1,\ldots,MB_{d_s}) \mid \\ (\forall\, r \,:\, 1 \le r \le d_s \,:\, (r \in NAL \;\Rightarrow\; MB_r = LB_r) \;\wedge \\ (r \in \{1,\ldots,d_s\} \setminus NAL \;\Rightarrow\; MB_r \in \{LB_r, UB_r\}))\} \end{array}$$

Proof.
According to the two conjuncts, we prove the lemma in two parts:

Part 1: $(\forall\, r \,:\, r \in NAL \,:\, MB_r = LB_r)$ for minimal $\tau_s(MB_1,\ldots,MB_{d_s})$.
We prove by contradiction.

$$\begin{array}{ll}
 & (\exists\, r \,:\, r \in NAL \,:\, (\exists\, \overline{MB}_r \,:\, \overline{MB}_r > LB_r \,: \\
 & \tau_s(MB_1,\ldots,\overline{MB}_r,\ldots,MB_{d_s}) \prec \tau_s(MB_1,\ldots,LB_r,\ldots,MB_{d_s}))) \\
\Rightarrow & \quad \{ \prec \text{ is total } \} \\
 & (\exists\, r \,:\, r \in NAL \,:\, (\exists\, \overline{MB}_r \,:\, \overline{MB}_r > LB_r \,: \\
 & \neg(\tau_s(MB_1,\ldots,LB_r,\ldots,MB_{d_s}) \preceq \tau_s(MB_1,\ldots,\overline{MB}_r,\ldots,MB_{d_s})))) \\
\Rightarrow & \quad \{ \text{ Requirement 2 } \} \\
 & (\exists\, r \,:\, r \in NAL \,:\, (\exists\, \overline{MB}_r \,:\, \overline{MB}_r > LB_r \,:\, \neg(LB_r \le \overline{MB}_r))) \\
\Rightarrow & \quad \{ \le \text{ is total } \} \\
 & (\exists\, r \,:\, r \in NAL \,:\, (\exists\, \overline{MB}_r \,:\, \overline{MB}_r > LB_r \,:\, (\overline{MB}_r < LB_r))) \\
\Rightarrow & \quad \{ \text{ contradiction to the assumption } \} \\
 & \mathit{ff}
\end{array}$$

Part 2: $(\forall\, r \,:\, r \in \{1,\ldots,d_s\} \setminus NAL \,:\, MB_r \in \{LB_r, UB_r\})$ for minimal $\tau_s(MB_1,\ldots,MB_{d_s})$. We prove by contradiction.

$$\begin{array}{ll}
 & (\exists\, r \,:\, r \in \{1,\ldots,d_s\} \setminus NAL \,:\, (\exists\, \overline{MB}_r \,:\, LB_r < \overline{MB}_r < UB_r \,: \\
 & \tau_s(MB_1,\ldots,\overline{MB}_r,\ldots,MB_{d_s}) \prec \tau_s(MB_1,\ldots,LB_r,\ldots,MB_{d_s}) \;\wedge \\
 & \tau_s(MB_1,\ldots,\overline{MB}_r,\ldots,MB_{d_s}) \prec \tau_s(MB_1,\ldots,UB_r,\ldots,MB_{d_s}))) \\
\Rightarrow & \quad \{ \text{ definition of the lexicographical order } \prec\text{; let } l \text{ be the first dimension} \\
 & \quad\;\; \text{where the values of } \tau_s(\ldots) \text{ are different and } c_{lk} \,:\, 0 \le k \le d_s \text{ be the} \\
 & \quad\;\; \text{coefficients in the } l\text{-th dimension of } \tau_s \; \} \\
 & (\exists\, r \,:\, r \in \{1,\ldots,d_s\} \setminus NAL \,:\, (\exists\, \overline{MB}_r \,:\, LB_r < \overline{MB}_r < UB_r \,: \\
 & \qquad\quad c_{l1}MB_1 + \cdots + c_{lr}\overline{MB}_r + \cdots + c_{ld_s}MB_{d_s} + c_{l0} \\
 & \qquad < \; c_{l1}MB_1 + \cdots + c_{lr}LB_r + \cdots + c_{ld_s}MB_{d_s} + c_{l0} \\
 & \quad \wedge \quad c_{l1}MB_1 + \cdots + c_{lr}\overline{MB}_r + \cdots + c_{ld_s}MB_{d_s} + c_{l0} \\
 & \qquad < \; c_{l1}MB_1 + \cdots + c_{lr}UB_r + \cdots + c_{ld_s}MB_{d_s} + c_{l0})) \\
\Rightarrow & \quad \{ \text{ arithmetic } \}
\end{array}$$

$(\exists\, r \,:\, r \in \{1,\ldots,d_s\} \setminus NAL \,:\, (\exists\, \overline{MB}_r \,:\, LB_r < \overline{MB}_r < UB_r \,:$
$c_{lr}\overline{MB}_r < c_{lr}LB_r \ \wedge\ c_{lr}\overline{MB}_r < c_{lr}UB_r))$
$\Rightarrow$ { range of $\overline{MB}_r$ }
$(\exists\, r \,:\, r \in \{1,\ldots,d_s\} \setminus NAL \,:\, (\exists\, \overline{MB}_r \,:\, LB_r < \overline{MB}_r < UB_r \,:$
$c_{lr} < 0 \ \wedge\ c_{lr} > 0))$
$\Rightarrow$ { arithmetic }
ff

The time for the first operation of a statement in the body of a loop nest with non-affine loops still depends on the lower and upper bounds of affine for loops but only on the lower bounds of non-affine loops.
To find this minimum we need Requirement 1. It ensures that the expression for the minimum consists only of affine subexpressions. Thus, we can apply existing methods (like [6], [8] or [27]) for finding the minimum.
The next step is to show that our time loops are running long enough. In the proof we apply the following lemma.

Lemma 39 (Positive Coefficients).
Let τ_s a d_s^t-dimensional affine schedule for statement s and $r : 1 \le r \le d_s$ a non-affine dimension. Then the r-th column $\vec{c_r}$ in the coefficient matrix of τ_s contains only values that are greater or equal to 0, i.e.,

$$(\forall\, r, \tau_s \,:\, r \in NAL \,:$$
$$\tau_s \textit{ satisfies Requirement 2} \ \Leftrightarrow\ \vec{c_r} \geq \vec{0})$$

Proof.
Let $x = (x_1,\ldots,x_r,\ldots,x_{d_s}) \in \mathcal{X}_s$ and $x' = (x_1,\ldots,x'_r,\ldots,x_{d_s}) \in \mathcal{X}_s$.

$x_r < x'_r$
$\Leftrightarrow$ { Requirement 2 }
$\tau_s(x) \preceq \tau_s(x')$
$\Leftrightarrow$ { τ_s is an affine function }
$\vec{c_0} + \vec{c_1}\, x_1 + \ldots + \vec{c_r}\, x_r + \ldots + \vec{c_{d_s}}\, x_{d_s}$
$\preceq \vec{c_0} + \vec{c_1}\, x_1 + \ldots + \vec{c_r}\, x'_r + \ldots + \vec{c_{d_s}}\, x_{d_s}$
$\Leftrightarrow$ { lexicographical order }
$\vec{c_r}\, x_r \le \vec{c_r}\, x'_r$
$\Leftrightarrow$ { arithmetic }
$\vec{c_r}\, (x_r - x'_r) \le \vec{0}$
$\Leftrightarrow$ { arithmetic }
$\vec{c_r} \geq \vec{0}$

Lemma 40 (Enumerating Enough Time).
Let s and τ_s as usual. The time at which an operation of s is executed is never later then the maximum value of the schedule with respect to the execution space known at that time step. Formally:

$$\begin{gathered}(\forall\, x\ :\ x=(i_1,\ldots,i_{d_s})\in\mathcal{X}_s\ :\ (\forall\, l\ :\ l\in\{1,\ldots,d_s\}\ :\\ \tau_s(x)\preceq\max\{\tau_s(MB_1,\ldots,MB_{d_s})\mid(l\in NAL\ \Rightarrow\ MB_l=max\_ub_l(\tau_s(x)))\ \wedge\\ (l\in\{1,\ldots,d_s\}\setminus NAL\ \Rightarrow\ MB_l\in\{LB_l,UB_l\})))\end{gathered}$$

Proof.
Let $\vec{c_l}$, $0\le l\le d_s$, the l-th column in the coefficient matrix of τ_s. Lemma 36 implies

$$\begin{array}{ll}
 & (\forall\, x\ :\ x=(i_1,\ldots,i_{d_s})\in\mathcal{X}_s\ :\ (\forall\, r\ :\ r\in NAL\ :\ i_r\le max_ub_r(\tau_s(x))))\\
\Rightarrow & \{\text{ Lemma 39 }\}\\
 & (\forall\, x\ :\ x=(i_1,\ldots,i_{d_s})\in\mathcal{X}_s\ :\ (\forall\, r\ :\ r\in NAL\ :\vec{c_r}\, i_r\le\vec{c_r}\, max_ub_r(\tau_s(x))))\\
\Rightarrow & \{\text{ let } \overline{MB}_k\in\{LB_k,UB_k\} \text{ the values that contribute to the maximum }\}\\
 & (\forall\, x\ :\ x=(i_1,\ldots,i_{d_s})\in\mathcal{X}_s\ :\\
 & \quad(\forall\, r\ :\ r\in NAL\ :\ \vec{c_r}\, i_r\le\vec{c_r}\, max_ub_r(\tau_s(x)))\ \wedge\\
 & \quad(\forall\, k\ :\ k\in\{1,\ldots,d_s\}\setminus NAL\ :\ \vec{c_k}\, i_k\le\vec{c_k}\,\overline{MB}_k))\\
\Rightarrow & \{\text{ arithmetic }\}\\
 & (\forall\, x\ :\ x=(i_1,\ldots,i_{d_s})\in\mathcal{X}_s\ :\\
 & \quad(\forall\, r\ :\ r\in NAL\ :\ \sum_r\vec{c_r}\, i_r\le\sum_r\vec{c_r}\, max_ub_r(\tau_s(x)))\ \wedge\\
 & \quad(\forall\, k\ :\ k\in\{1,\ldots,d_s\}\setminus NAL\ :\ \sum_k\vec{c_k}\, i_k\le\sum_k\vec{c_k}\,\overline{MB}_k))\\
\Rightarrow & \{\text{ arithmetic }\}\\
 & (\forall\, x\ :\ x=(i_1,\ldots,i_{d_s})\in\mathcal{X}_s\ :\ (\forall\, r,k\ :\ r\in NAL,\ k\in\{1,\ldots,d_s\}\setminus NAL\ :\\
 & \quad\sum_r\vec{c_r}\, i_r+\sum_k\vec{c_k}\, i_k\ \le\ \sum_r\vec{c_r}\, max_ub_r(\tau_s(x))+\sum_k\vec{c_k}\,\overline{MB}_k))\\
\Rightarrow & \{\text{ arithmetic }\}\\
 & (\forall\, x\ :\ x=(i_1,\ldots,i_{d_s})\in\mathcal{X}_s\ :\\
 & \quad(\forall\, l,r,k\ :\ l\in\{1,\ldots,d_s\},\ r\in NAL,\ k\in\{1,\ldots,d_s\}\setminus NAL\ :\\
 & \qquad\sum_l\vec{c_l}\, i_l\ \le\ \sum_r\vec{c_r}\, max_ub_r(\tau_s(x))+\sum_k\vec{c_k}\,\overline{MB}_k))\\
\Rightarrow & \{\text{ arithmetic }\}\\
 & (\forall\, x\ :\ x=(i_1,\ldots,i_{d_s})\in\mathcal{X}_s\ :\\
 & \quad(\forall\, l,r,k\ :\ l\in\{1,\ldots,d_s\},\ r\in NAL,\ k\in\{1,\ldots,d_s\}\setminus NAL\ :\\
 & \qquad\sum_l\vec{c_l}\, i_l+\vec{c_0}\le\ \sum_r\vec{c_r}\, max_ub_r(\tau_s(x))+\sum_k\vec{c_k}\,\overline{MB}_k+\vec{c_0}))\\
\Rightarrow & \{\text{ definition of the } \vec{c_l} \text{ and ``}\le\text{ implies }\preceq\text{'' }\}\\
 & (\forall\, x\ :\ x=(i_1,\ldots,i_{d_s})\in\mathcal{X}_s\ :\ (\forall\, l\ :\ l\in\{1,\ldots,d_s\}\ :\\
 & \quad(l\in NAL\ \Rightarrow\ MB_l=max_ub_l(\tau_s(x)))\quad\wedge\\
 & \quad(l\in\{1,\ldots,d_s\}\setminus NAL\ \Rightarrow\ MB_l=\overline{MB}_l)\ \wedge\\
 & \quad(\tau_s(x)\ \preceq\ \tau_s(MB_1,\ldots,MB_{d_s}))))
\end{array}$$

$$\begin{array}{ll} \Rightarrow & \{\ \text{definition of } \overline{MB}_l \text{ and analogous application of Part 2 of the proof} \\ & \quad \text{of Lemma 38, where } \prec, < \text{ and } > \text{ are replaced by } \succ, > \text{ and } <, \\ & \quad \text{respectively}\ \} \end{array}$$

$$\begin{aligned} &(\forall\, x : x = (i_1, \ldots, i_{d_s}) \in \mathcal{X}_s : (\forall\, l : l \in \{1, \ldots, d_s\} : \\ &\quad \tau_s(x) \preceq \max\{\tau_s(MB_1, \ldots, MB_{d_s}) \mid (l \in NAL \;\Rightarrow\; MB_l = max\_ub_l(\tau_s(x))) \\ &\qquad\qquad\qquad\qquad \wedge\ (l \in \{1, \ldots, d_s\} \setminus NAL \;\Rightarrow\; MB_l \in \{LB_k, UB_k\}))) \end{aligned}$$

Lemma 41 (Termination of Time Loops).
The time loops terminate after a finite number of steps if the source loops terminate, i.e., after some time step $\bar{t}$ the bounds max_ub_r of non-affine loops do not change anymore:

$$(\exists\, \bar{t} : \bar{t} \geq min\_t_s : (\forall\, t : t > \bar{t} : max\_ub_r(t) = max\_ub_r(\bar{t})))$$

Proof. (Sketch)
We prove this assertion informally.

$$\begin{aligned} max\_ub_r(t) &= \max\{UB_r(x_1, \ldots, x_r) \mid \tau_s(x_1, \ldots, x_{d_s}) \leq t\} \\ max\_ub_r(\bar{t}) &= \max\{UB_r(x_1, \ldots, x_r) \mid \tau_s(x_1, \ldots, x_{d_s}) \leq \bar{t}\} \end{aligned}$$

If the two sets contain the same elements, both maxima are sure to be equal. What sets are possible? The number of elements in the set of all different upper bounds $\{UB_r(i_1, \ldots, i_r) \mid LB_k \leq i_k \leq UB_k \;\wedge\; 1 \leq k \leq r\}$ is finite if all surrounding loops terminate. This means that there has to be an instant of time ($\bar{t}$) where all different upper bounds UB_r are considered. For all $t \geq \hat{t}$, the set of upper bounds does not change and so the maximum of this set does not change, either.
Since the max_ub_r are the only values in the expression for the maximum of time that change over time and there is a point in time at which this values do not change anymore, the loops that enumerate the time dimension must terminate after a finite number of time steps.

Lemma 41 means that, if we replace the original upper bound ub_r of a loop by max_ub_r, we can find bounds for the time loops of the target program. However, the polyhedron model is based on solving systems of linear inequalities — and max_ub_r is not a linear expression. For the parallelization steps of dependence analysis and computing schedules and allocations for the operations of the statements in a source program, we can replace the new upper bound of non-affine loops by a structure parameter that represents a

"large enough" value[10] (Requirement 3). A constant parameter is an affine expression, and so all parallelization steps can be carried out as usual.
When generating the target code, we have to replace this new inserted artificial parameter by its 'proper' value. In the expressions of the target loop bounds this is max_ub_r. The previous proofs and explanations show that, at any given time, this value is large enough. We only have to take care not to read values of max_ub_r at a time at which no operation of a statement in the body of the loop is executed, i.e., max_ub_r must contain a valid value. This problem can be solved by holding a (global) flag $valid\_ub_r$ that is initialized with *ff* and is set to *tt* by the processor that is assigning to max_ub_r for the first time. We implement the administration of the flag $valid\_ub_r$ as a side effect of the loop instructions (see Chapter 4 and especially Section 4.2).

The set of points that are to be executed in the same time step is called a *time slice*. Lemma 41 implies that there is a finite number of such time slices of a statement.

Remark. Every single time slice of a statement contains a finite number of operations.

As all for loops terminate[11], they enumerate a finite number of operations. Thus, the number of operations executed at a given time step must be finite, too. For while loops the finiteness of time slices is proved in [11].

Now we know that, at a given instant of time, we only have to address a finite number of processors. The following lemmata ensure that we can describe this set of processors and that it is large enough.

Lemma 42 (Lower Bounds for Space Loops).
The lower bounds of the space loops can be computed as in the polytope model.

Proof.
Let $\vec{LB}_s = \begin{pmatrix} LB_1 \\ \vdots \\ LB_{d_s} \end{pmatrix}$ the d_s-dimensional vector of the lower bounds of the loops surrounding statement s and $p = (p_1, \ldots, p_{d_s^p})$ the values of the space coordinates. For a point in $\mathcal{TS}_s$ at a given time step $\bar{t} = (\bar{t}_1, \ldots, \bar{t}_{d_s^t})$, the following condition must be satisfied:

$$\vec{LB}_s \leq T_s^{-1}(\bar{t}, p) \tag{3.1}$$

[10] It is assumed to be positive if the stride ST_r is positive and negative if the stride is negative.
[11] PASCAL-like loop semantics.

where each row in T_s^{-1} represents a source index corresponding to the respective lower bound.
As the expressions in $\vec{LB}_s$ are all affine and T_s^{-1} is also an affine function, this system of inequalities can be solved to express p_1 in dependence on $\bar{t}$, p_2 in dependence on $\bar{t}$ and p_1, etc. This can be done by applying one of the algorithms in [6], [8] or [27].

We have to show that we can also find the other bound of the range of the space indices.

Lemma 43 (Upper Bounds for Space Loops).
The upper bounds of the space loops can be computed by allocators based on the polytope model.

Proof (Sketch).
Let $\vec{UB}_s = \begin{pmatrix} UB_1 \\ \vdots \\ UB_{d_s} \end{pmatrix}$ the d_s-dimensional vector of the upper bounds of the loops surrounding statement s and $p = (p_1, \ldots, p_{d_s^p})$ the values of the space coordinates. For a point in $\mathcal{TS}_s$ at a given time step $\bar{t} = (\bar{t}_1, \ldots, \bar{t}_{d_s^t})$ the following condition must be satisfied:

$$T_s^{-1}(\bar{t}, p) \quad \leq \quad \vec{UB}_s \tag{3.2}$$

where each row in T_s^{-1} represents a source index corresponding to the respective lower bound.
Again T_s^{-1} is an affine function, but now the expressions for UB_r, where $1 \leq r \leq d_s$ and $r \in NAL$, are given by max_ub_r and these are non-affine. Therefore, in this form the inequalities cannot be solved by the algorithms used for the the lower bounds.
If we take a look at Lemma 36, we notice that the values of max_ub_r form a function of time. As in Lemma 42, in our synchronous view, time is fixed for the space dimensions and so the values of UB_r can be considered constant by the allocator. Constants are affine and so the algorithms for calculating the lower bounds can be applied.
When generating the target code, we only have to replace this artificial constant by the real value of max_ub_r.

Remark. The same arguments would hold for Lemma 42, too, i.e., Requirement 1 would not be necessary for allocations, if we introduced a min_ub_r for non-affine lower bounds. However, for finding lower time bounds, we need the lower source bounds to be affine (and Requirement 2).

In Lemmata 42 and 43, we state which bounds for the space loops we want to use. However, we modified the source loop bounds and so we have to show that we are enumerating enough (virtual) processors.

Lemma 44 (Enumerating enough Space).
Let $s \in \mathcal{S}_\mathcal{P}$ a statement in the source program $\mathcal{P}$ and T_s the transformation for s. Then:

$$(\forall\, x \,:\, x \in \mathcal{X}_s \,: \\ T_s(x) = (\bar{t}, p) \textit{ satisfies conditions 3.1 and 3.2})$$

Proof.
$T_s^{-1}(T_s(x)) = x$, as we only consider bijective transformations. We have to show $(\forall\, x \,:\, x = (i_1, \ldots, i_{d_s}) \in \mathcal{X}_s \,:\, x$ satisfies Conditions 3.1 and 3.2).
Condition 3.1 is trivially satisfied, since the definition of for loops and while loops implies that $(\forall\, r \,:\, 1 \leq r \leq d_s \,:\, LB_r \leq i_r)$ and we do not change the lower bounds.
Since we have changed the upper bounds of non-affine loops, we have to take a closer look at Condition 3.2.

Case 1: i_r is the index of an affine for loop

$$\begin{array}{ll} & is\_for(r) \\ \Rightarrow & \{ \text{ semantics of for loops } \} \\ & i_r \leq UB_r \end{array}$$

Case 2: i_r is the index of a non-affine loop

$$\begin{array}{ll} & r \in NAL \\ \Rightarrow & \{ \text{ we changed the upper bound } \} \\ & i_r \leq max\_ub_r \\ \Rightarrow & \{ \text{ Lemma 36 } \} \\ & i_r \leq max\_ub_r(\tau_s(x)) \end{array}$$

I.e., at a given point in time, we enumerate enough processors. Since $UB_r \leq max\_ub_r$, in general we enumerate too many.

3.3.4 Different Methods for Termination Detection

The basis of the termination detection described in this thesis is the calculation of the maximum of upper bounds, max_ub_r. In [1] and [11] two other methods are presented: the *signaling scheme* [11] for distributed-memory computers and the *counter scheme* [1] for shared-memory computers. Their goal is to avoid as many communications and overhead as possible (within the scope of the respective machine model).

In our thesis we concentrate on the "synchronous parallelism on a shared-memory machine" point of view, since this view is closest to the usual sequential way of thinking and since we wanted to learn about parallel execution at an abstract level. Further, in [5] methods for mapping parallel shared-memory programs to distributed-memory machines are described. These are also implemented in LooPo [13]; so the decision for generating shared-memory programs does not restrict the choice of target machine. However, we expect that, for distributed-memory machines, the signaling scheme yields better results than the adapted methods for shared-memory machines. Anyway, we think that many of the theoretical results in this thesis are valid for the signaling scheme, too.

In the present section, we compare the counter scheme [1] to the way we describe termination detection in this thesis, which we call the *maximum scheme*.

In the counter scheme, a global shared counter is incremented before a new 'tooth' (an instance of a loop) starts (in any arbitrary dimension) and is decremented when a tooth terminates. The program terminates when every tooth has terminated, which is the case when the counter is 0 again. [1] also describes possible optimizations to prevent the counter from becoming a bottleneck.

The main advantage of the counter scheme is the reduction of synchronization by one dimension. The management of the counter does not affect the innermost dimension which causes a considerably decrease in run time overhead; the more iterations the instances of the innermost loop execute, the bigger the gain.

To generalize the counter scheme for the execution of non-perfect general loop nests, essentially for every tooth of a `for` loop (that is not the innermost loop) the counter must be incremented by the number of iterations the loop will execute. This is the number of teeth of the next inner loop which will start into the next dimension. When an instance of a `for` loop terminates, the counter has to be decremented by 1 in the same way as for `while` loops.

However, the target execution space is not the same for all statements that have the same source execution space, so it is not possible to say precisely "when an instance of a loop terminates". Consider two statements with the same dimensionality r in the body of a loop nest. An instance of the surrounding loop terminates only when all operations of the statement whose schedule yields the largest values have been executed. It must be ensured that this statement decrements the counter (as side effect). We suspect that, for non-perfect loop nests where every statement has its own transformation, it is hard to find this statement at compile time; but we refer to possible future studies on this.

Another drawback is the reduced information the counter scheme stores. Only the number of currently executing loop instances is known and that leads to a very inexact scanning of the space dimensions. As partitioning (see also Section 3.5) divides a space dimension into a space dimension for the really existing processors and a time dimension, this rough approximation costs also time.

A generalization of the counter scheme has to store larger numbers than the maximum scheme. The latter stores only the maximum of the upper bounds of non-affine loops, whereas the former stores the number of instances of all kinds of loops. This might cause practical problems.

The transformation of the counter scheme for distributed-memory machines causes a lot of time-consuming communication which can be used to calculate the max_ub_r as well. This provides more information and makes an explicit counter unnecessary.

We illustrate the differences between the spaces scanned by the counter and the maximum scheme by means of the following figures. Consider the spaces shown in Figure 3.4.

The left part shows the source index space of statement s, e.g., given by the source program of Section 3.1. The right part shows the target index space determined by the transformation $T_s = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$.

Note that the j-dimension of the source index space has no upper bound in the polyhedron model. Accordingly, both dimensions of the target index space have no upper bounds, either (expressed by the dark grey color shading off into white).

The following three figures show different sample target execution spaces with their respective scanned target spaces.

For Figure 3.5 the values of $a[i]$ happen to be $5, 3, 2, 2, 1, 1$. We notice that the counter scheme enumerates one more (virtual) processor than the maximum scheme, but the latter enumerates considerably more time.

The region containing the horizontal dashed lines contains the points which would be scanned additionally by the maximum scheme if the target index

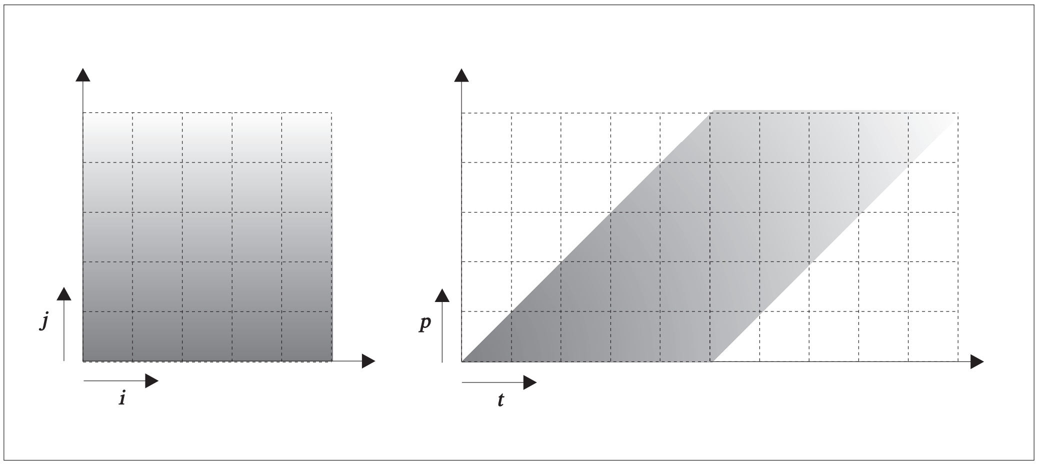

Figure 3.4: The index spaces

space (known at compile time) was not considered. Since our maximum scheme considers the intersection of the target space and the space that would be scanned if only the *max_ub* values were used (without concerning the index space), the dashed region is cut off the scanned target space.
This is an example where the difference between both schemes becomes very clear and the counter scheme delivers better results. The reason is that the first tooth is the longest and the upper bound of j is monotonously decreasing.
Figure 3.6 shows a case that is more common. The values of $a[i]$ are 1, 1, 3, 2, 4, 1.
Because the longest tooth is close to the end, the maximum scheme is about as good as the counter scheme; it only needs one additional time step. The space dimension is scanned more precisely by the maximum scheme. The additional information given by the *max_ub* values causes the 'jagged', thus more precise, left border of the target space scanned by the maximum scheme. Again the dashed region is actually not scanned.
Figure 3.7 depicts a target space that is supported best by the maximum scheme. Both methods yield minimal possible time, but only the maximum scheme scans the space dimension precisely. We can say that the maximum scheme yields the same run time as the counter scheme if the upper bound of the j-dimension is increasing monotonically. Then the maximum scheme, in general, scans a smaller subset of the target index space than the counter scheme.

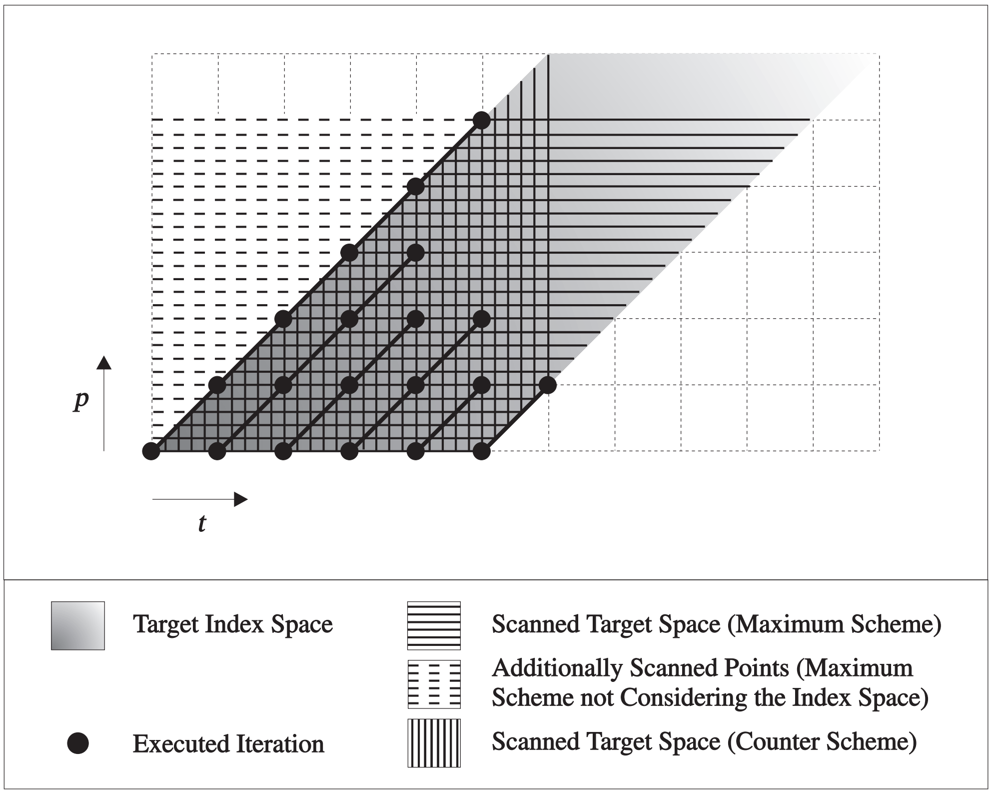

Figure 3.5: The scanned target spaces: counter vs. maximum

Summary of the results:

- Both schemes depend heavily on the actual target execution space. If there is statistical knowledge about the values occurring at run time then it is possible to decide which scheme to use.

  In general: monotonically increasing upper bounds are supported better by the maximum scheme whereas with virtual processors the time dimensions are scanned more precisely by the counter scheme.

- The usage of a greater number of real processors has a greater effect with the counter scheme than with the maximum scheme. However, naturally the latter one is improved, too.

- No method can be said to be "best", but a combination of both methods is better than any of the two alone. It scans the intersection of both

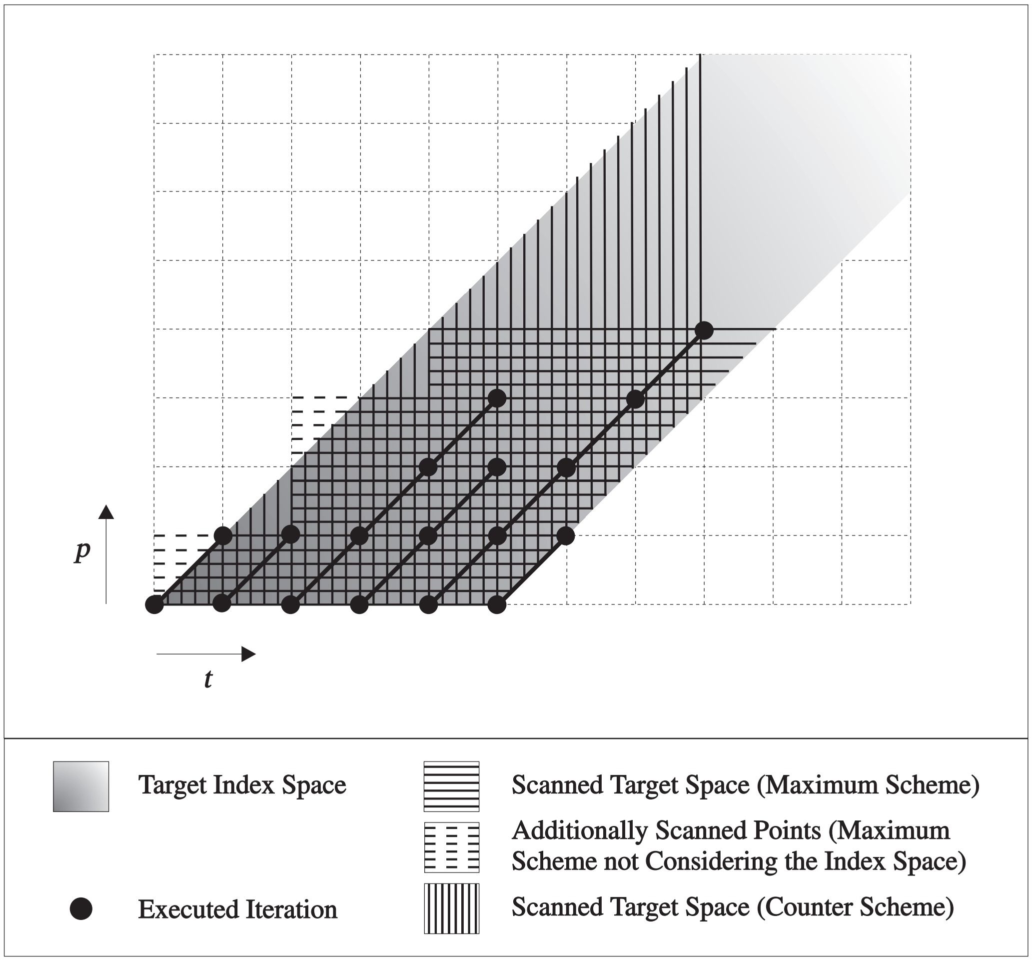

Figure 3.6: The scanned target spaces: counter vs. maximum

scanned target spaces and this is smaller than any individual scanned target space.

We choose to use the maximum scheme for termination detection, because it constitutes a more consequent generalization of the existing method for affine for loops. Additionally we avoid the necessity of finding that statement among all statements with the same execution space whose schedule yields the largest values.
It may turn out that the biggest (practical) advantage of the maximum scheme is the comparatively small storage overhead for the values of max_ub_r. Further studies will have to deliver more concrete and measurable results.

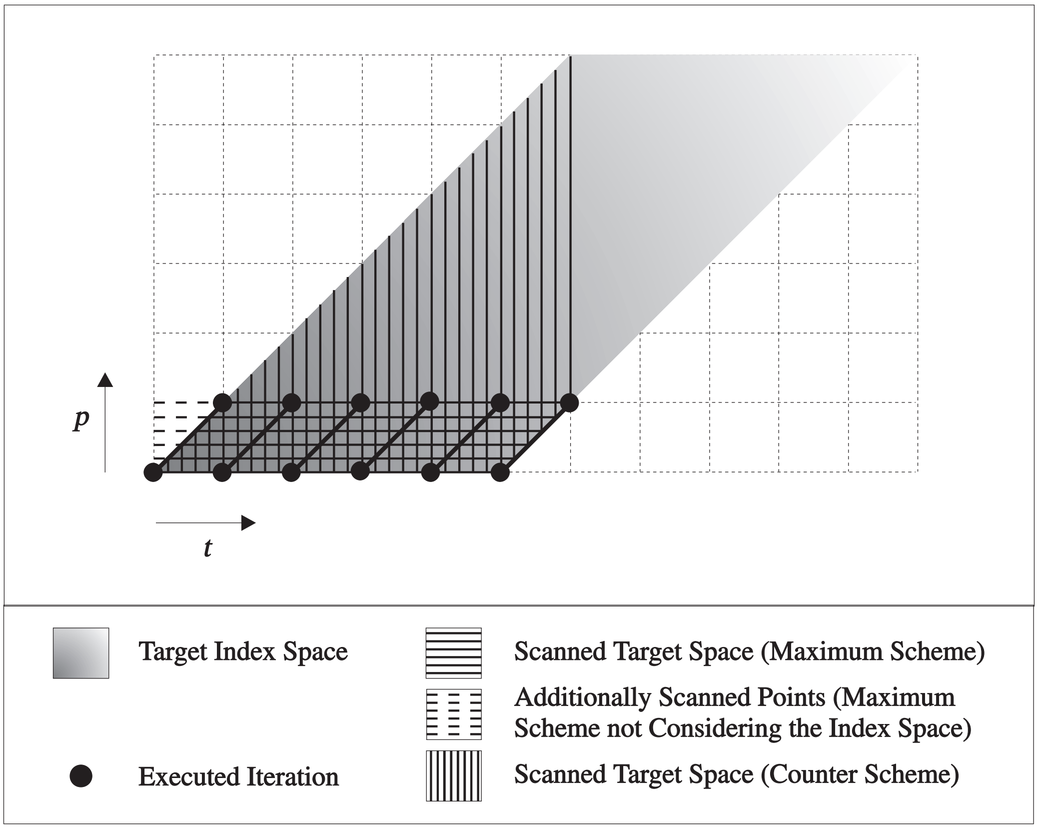

Figure 3.7: The scanned target spaces: counter vs. maximum

3.4 "By-Statement View"

All our deliberations so far have only been concerned with perfect loop nests with only one body statement, or rather with a body that is subject to one and the same transformation function for all statements. This is how the scheduling and allocation method by Lamport [18, 26] can treat bodies of perfectly nested loops. Thus, all operations of different statements at the same iteration are mapped to one instant in time and one processor, and the whole body of one iteration is executed sequentially. Only different iterations are executed in parallel.
Modern scheduling and allocating techniques [2, 4, 7, 21, 26] calculate an affine schedule and allocation for each individual statement. This implies that operations of different statements and different iterations can be executed in parallel and that loops do not have to be perfectly nested.
The disadvantage is that, in general, every statement has a different target

execution space even if the source execution spaces are equal. These target execution spaces must be scanned by one common target loop nest, i.e., they must be merged.

Wetzel [25] offers a method for merging target execution spaces of statements in the body of loop nests with only affine loops.
Since we base our code generation technique on the output (created by the so-called "synchronous run time method") of Wetzel's code generator, we briefly explain its basics here without considering special cases.

Roughly, the transformation of the source execution spaces yields as many d-dimensional target polytopes as there are statements (say n) in the source program. One dimension of the union of all these n target polytopes is enumerated by one single target loop, a sequential one if it is a time dimension and a parallel one if it is a space dimension. The lower bound of the loop for some target dimension is determined by the minimum of the n minima of the respective dimension. Analogously, the upper bounds are determined by the "maximum of all maxima".

Example 10 (Merging the Target Execution Spaces).
Consider the following source program:

```
for i := 0 to 5
   for j := 0 to 2
      s1
      s2
   end
end
```

Let $T_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, $T_2 = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$, $T_1^{-1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, $T_2^{-1} = \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}$ be the transformation functions and their inverses of statement s_1 and s_2, respectively. The two target execution spaces are depicted in Figure 3.8 (left) and can be described by the following inequalities:

Statement 1	Statement 2
$0 \leq t \leq 5$	$0 \leq t \leq 5$
$0 \leq p \leq 2$	$t \leq p \leq t+1$

Next we have to compute the minimum and maximum of every dimension for every statement. This is very simple in this example:

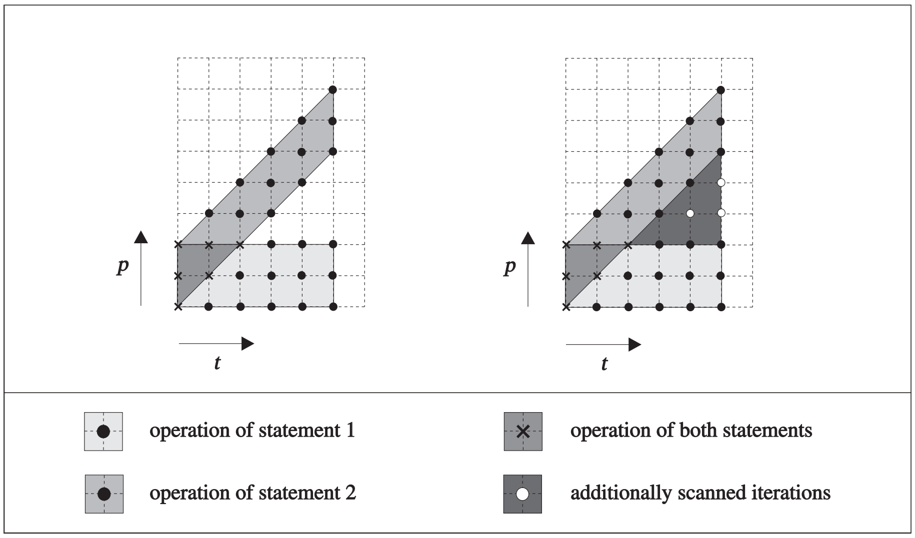

Figure 3.8: Target execution spaces (left) and scanned execution space (right)

Dimension	Statement 1		Statement 2	
	min	max	min	max
t	0	5	0	5
p	0	2	t	$t+2$

The last step is to compute the minimum and maximum of t and p over all statements. These values are the new target loop bounds:

$$\begin{aligned} \min(0,0) &\leq t \leq \max(5,5) \\ \min(0,t) &\leq p \leq \max(2,t+2) \end{aligned}$$

The union of all grey regions in Figure 3.8 represents the scanned target space. It is not exactly the union of the two execution spaces. Thus, there are points at which no, one or two operations have to be executed, e.g., $(4,3)$, $(3,1)$, $(1,2)$, respectively.
To prevent the execution of points that are not in the execution space of a statement, every statement in the target program is guarded by its transformed predicate *executed*.

Note the two different levels at which an inexact scanning of the target execution spaces can occur:

1. As described in Sections 3.2 and 3.3 and as considered in Chapter 2, the target execution space *of a statement* may not be precisely scannable.

2. In this section we see that the union of the target execution spaces of different statements is not precisely scannable — even if the individual target execution spaces were precisely scannable.

As predicate *executed* of a statement describes its execution space exactly, it solves both problems at one go.
The execution predicates here are the same for both statements, since both statements are in the same loop nest and have the same level (namely 2):

$$\begin{array}{c} executed_{s_1}(i,j) \;\equiv\; executed_{s_2}(i,j) \;\equiv \\ 0 \le i \le 5 \;\wedge\; (i-0)\%1 = 0 \;\wedge\; 0 \le j \le 2 \;\wedge\; (j-0)\%1 = 0 \end{array}$$

Since every statement has its own transformation, we get two predicates *executed* on the target side, one for each statement:

Statement 1	Statement 2
$targ\text{-}executed_{s_1}(t,p) \equiv$ $0 \le t \le 5 \wedge (t-0)\%1 = 0 \wedge$ $0 \le p \le 2 \wedge (p-0)\%1 = 0$	$targ\text{-}executed_{s_2}(t,p) \equiv$ $0 \le t \le 5 \wedge (t-0)\%1 = 0 \wedge$ $t \le p \le t+2 \wedge (p-t-0)\%1 = 0$

With t as time and p as processor dimension we can construct the target program:

```
for t := min(0,0) to max(5,5)
   forall p := min(0,t) to max(0,t+2)

      if targ-executed_s1(t,p) then
         s'_1
      endif

      if targ-executed_s2(t,p) then
         s'_2
      endif

   end
end
```

s'_1 and s'_2 are the transformed statements, where i and j are expressed in terms of t and p.

This target program ensures the scanning of a large enough space and avoids execution of operations at inappropriate iterations.

Remark. The example illustrates only the basic steps of target code generation à la Wetzel. [25] describes a number of extensions, e.g., the handling of non-bijective and non-unimodular transformations, not mentioned here.

3.5 Implications of Finiteness

Example 3.1 illustrates that it is not always possible to generate an asynchronous parallel target program, because the upper bounds of the space loops may not be known at the beginning of the execution. On the other hand, there are machines that work asynchronously and consequently need asynchronous programs. The logical implication would be that, in general, only parallel synchronous or sequential asynchronous target programs are derivable.

However, this scenario does not take into account that we have only a finite number (say n) of processors and must apply a partitioning algorithm (e.g., [23]) to map the virtual processor coordinates onto real processors.

Essentially when partitioned, one virtual processor loop becomes a nest of two loops. One of them enumerates all n processors in some dimension and the other counts the number of steps each of the n processors has to make in order to enumerate the whole (virtual) dimension.

Example 11 (Partitioning).
Let the program

$$\mathsf{for}\ p := 0\ \mathsf{while}\ \mathit{cond}\ \mathsf{do}\ \mathit{body}\ \mathsf{end}$$

enumerate a space dimension in the target program that could be executed in parallel if its upper bound were known. Further let n be the number of (real) processors in the respective dimension. Partitioning would change the loop as follows:

```
for pp := 0 to n − 1 do
    for pt := pp while cond step n do
        p = pp + pt
        body
    end
end
```

The while loop still does not offer the opportunity for more parallelism — but the new for loop does. Since all statements in the body may be carried out in parallel, we can change the for loop into a forall loop and exploit the maximum parallelism our machine offers. The second loop would be sequential in any case (also if it was a for loop), because we do not have more than n processors in the respective dimension. Thus, no more parallelism is possible.
With a finite number of processors we can have partially parallel while loops with a speed-up of n.

The theoretically interesting conclusion of this section is that, in practice, we can also derive asynchronous target programs that exploit the maximum parallelism that is offered by a given multi-processor machine. It does not exploit the maximum theoretical parallelism offered by the source program.

Chapter 4

Implementation in LooPo

LooPo [13] is a prototype of an automatic parallelizer for sequential source programs which are written in a loop language. It offers a variety of methods for dependence analyses, for finding (piecewise affine by-statement) schedules and allocations and for generating synchronous or asynchronous target programs for distributed-memory or shared-memory machines. It also provides a graphical tool to view the source and target execution spaces of the different statements.
At present, LooPo can deal with (possibly 0-dimensional) arrays as data structures and affine for loops as control structures. Within the scope of this diploma thesis we implement an extension to LooPo which makes it capable of dealing with if statements and all types of for and while loops.

In Chapter 3 we described the theoretical basis of our method. However, for historical reasons the current implementation of LooPo lacks structures that are general enough to express programs with general loop nests: since the execution conditions of affine for loops can be evaluated at compile time, there was no need to find schedules and allocations for loop instructions so far.
The execution condition of a general loop, however, must be evaluated at run time. In Chapters 2 and 3 we saw that a loop instruction consists of several substatements whose results are part of the respective execution condition. We have to find schedules and allocations for these substatements, to be able to evaluate the execution condition at run time. Therefore, and to meet the needs of the parallelization tools like the dependence analysis, scheduler, allocator and the present code generator that we want to use as far as possible, we have to transform the source program.
This program transformation (at the source level) is called *normalization*, because it puts all loops in a similar form. Note that our normalization —

in contrast to other normalizations that can be found in the literature, e.g., [20] — does, in general, not preserve the semantics of the source program: the execution spaces of statements are changed. We will also transform if statements and insert new statements, so that the program can be handled by the "other tools". Since normalization does not preserve the semantics, we have to retransform the program during code generation, i.e., we have to restore the original execution spaces.

4.1 Normalization

We describe the normalization in five steps. These are implemented in the normalization module (see Figure 0.2).

Input: a program as described in Chapter 1, i.e., with all kind of loops and with if statements

Output:
- a program with different semantics, only affine for loops and no if statements, but with additional (user-defined control) dependences and
- a list of records that establish a relation between every statement and the execution conditions of its surrounding non-affine loops and if statements. This list also contains information of whether a statement was an ordinary statement in the source program or whether it originates from an if statement, from a while loop or from a non-affine for loop.

1. The for loops with a non-affine lower bound are normalized to have a lower bound that is 0, i.e., a for loop

   for i := lb to ub step ST do
       $body$
   end

   is changed to

   $LB[indlist] := lb$
   for i := 0 to $(ub - LB[indlist])$ step ST do
       $body'$
   end

where in *body'* all occurrences of i are replaced with $i{+}LB[indlist]$. The assignment before the for instruction is necessary, because some statement in the body may change values occurring in lb and a subsequent evaluation of $i+lb$ would yield the wrong result. Furthermore, this allows lower bounds with side effects. *indlist* is the list of all enclosing loop indices. The usage of an array ensures that no possible parallelism is destroyed.

Remark. *LB* must be unique within the source program.

With regard to parallelization, we have to ensure that the new assignment is executed before all the body statements. We can achieve this by inserting user-defined dependences from the new assignment to all statements in the body of the loop.

After this step, we have only for loops with affine lower bounds. Thus, requirement 1 in Chapter 3 is satisfied.

Note that this step preserves the semantics of the source program.

2. All for loops with a non-affine upper bound are changed to

```
boolean validfubb := ff
integer maxfubb := 0
...
UB[indlist] := ub
maxfubb := max(maxfubb, UB[indlist])
for i := lb to FUBB step ST do
   if (lb <= i <= UB[indlist] and (i − lb)%ST = 0) then
      body
   endif
end
```

If the stride is negative, then we have to replace the calculation of $maxfubb$ by $minfubb := min(minfubb, UB[indlist])$.

This step alters the semantics of the program. The upper bound will become greater than it was in the source program, i.e., the index space of the body is enlarged. During code generation, we must take care to obtain the correct execution space.

Again new dependences from both new assignments to every body statement have to be inserted.

maxfubb is the same as *max_ub* for a for dimension in the last chapter and computes the new upper bound of the loop. It is initialized with an arbitrary value. *validfubb* is a boolean flag that indicates whether a value of *maxfubb* is already written. If so, *maxfubb* does not contain an undefined value and may be read from now on. *validfubb* is initialized with *ff*.

FUBB (**F**or loop **U**pper **B**ound **B**lob) is a new constant parameter whose value has to be assumed to be infinite by the other tools. It must be replaced by the value of the respective *maxfubb* during code generation to ensure the termination of the target program. We cannot simply replace *FUBB* by *maxfubb*, because this would yield a non-affine bound which cannot be handled by the other tools.

Remark. The variables *maxfubb* and *validfubb* must be unique for every non-affine for loop within the source program.

3. We change while loops as described in Chapter 3:

   while *cond* do *body* end

   is changed to

```
integer maxwubb := 0
...
for new_i := 0 to WUBB do
   if (executed[indlist]) then
      body
   endif
end
```

   What applies to *FUBB* and *maxfubb* also applies to *WUBB* (**W**hile loop **U**pper **B**ound **B**lob) and *maxwubb*. The latter is declared and initialized here but is only introduced into the target program during retransformation (see Section 4.2). We do not need a *validwubb* for a while loop, because we know the lower bound. It is 0 for every instance of the loop and does not depend on any other values. Initializing *maxwubb* with 0 means that every normalized while loop executes at least one iteration, which is necessary to check the while condition at least once.

The single if statement is not powerful enough to ensure the correct execution space of the body. It has to check the execution condition of the while loop and this is more complex (see Section 2.1.2). We have to take care of this when we retransform the target program.

We still have to eliminate the if statements.

4. There are three kinds of if statements in the source program:

   (a) Standard if statements are transformed as follows:

   $$\textsf{if } cond \textsf{ then } body \textsf{ endif}$$

   is changed to

   $$new\_if[indlist] := cond$$
   $$body$$

   The value of the condition must be stored for the same reasons as the bounds of non-affine for loops.

   This transformation changes the execution space of the body, as it would be executed unconditionally this way. We must take care of that during code generation. Therefore we must record to which body every statement originally belonged.

   We insert dependences from the new assignment to every statement in the body of the original if statement.

   (b) If the if statement results from a while loop, we have to insert additionally the while dependence from an operation of the new assignment to the operation at the next iteration of *new_i*. The reason for this dependence is the calculation of *executed*. The new assignment (together with the new for loop) represents the while loop in the normalized source program.

   (c) An if statement originating from a non-affine for loop can only have a condition of the form:

   $$lb <= i <= UB[indlist] \textsf{ and } (i - lb)\%ST = 0 \text{ or}$$
   $$lb <= i <= (UB[indlist] - LB[indlist]) \textsf{ and } (i - lb)\%ST = 0$$

   As the values occurring in these expressions are already stored and never written again, we need not insert the new assignment. The dependences we inserted for the $UB[indlist]$ statement and $LB[indlist]$ statement already contain the control dependences from the if statement to its body. Note that this is just an optimization to save memory.

We must record the origins of if statements to be able to make the case distinctions described above.

Remark. The new inserted user-defined dependences embody the control dependences shown in Figures 2.2, 2.3, 2.4, 3.1 and 3.2.

We illustrate the normalization by means of the following example and return to this example again later to describe the retransformation during code generation.

Example 12 (Parallelization of Dynamic Execution Spaces).
Given the following source program, which contains an affine for loop, a non-affine for loop, an if statement, and a while loop and which is not perfectly nested, we apply the normalization steps "one by one" and "inside out", i.e., we begin with a loop or if statement at an innermost level, apply all five steps and continue with the next outer level.

```
    /* source program */

    for i := 0 to 4 do
      if (a[i] = 0) then
        for j := b[i] to b[i] + c[i] do
1:        b[i] := b[i] + 1
2:        c[i] := c[i] + 1
        end
      endif
      while (a[i] > 0) do
3:      a[i] := a[i] - 1
      end
    end
```

After applying the normalization steps we get the following result:

```
    /* normalized source program */

    boolean validfubb := ff
    integer maxfubb := 0
    integer maxwubb := 0

    for i := 0 to 4 do
7 :   new_if_1[i] := (a[i] = 0)
4 :     LB[i] := b[i]
5 :     UB[i] := abs(c[i])
6 :     maxfubb := max(maxfubb, UB[i])
        for j := 0 to FUBB do
1 :       b[j + LB[i]] := b[j + LB[i]] + 1
2 :       c[j + LB[i]] := c[j + LB[i]] + 1
        end
      for k := 0 to WUBB do
8 :     new_if_2[i, k] := (a[i] > 0)
3 :       a[i] := a[i] − 1
      end
    end
```

inserted dependences		
source	target	h-transformation
7	4 5 6	$\begin{pmatrix} 1 & 0 \end{pmatrix}$
4	1 2	$\begin{pmatrix} 1 & 0 & 0 \end{pmatrix}$
5	1 2	$\begin{pmatrix} 1 & 0 & 0 \end{pmatrix}$
6	1 2	$\begin{pmatrix} 1 & 0 & 0 \end{pmatrix}$
8	8	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \end{pmatrix}$
	3	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$

Note that we did not insert dependences $7\delta1$ and $7\delta2$, although statements 1

and 2 belong to the body of the if statement. This is legal, because we know that we insert, e.g., dependences $7\delta4$, $4\delta1$ and $4\delta2$ which contain dependences $7\delta1$ and $7\delta2$ in their transitive closure.
This optimization can be done because we know the normalization algorithm. It does not eliminate arbitrary transitive dependences in source programs. These are treated by the dependence analysis afterwards.

Remark. The inserted dependences describe only a temporal sequence and correspond to the control dependences shown in Figures 3.1, 3.2 and 3.3. It is still up to the dependence analysis to identify the data dependences.

As stated above, the normalization yields a relation, we call it STATEC (**STA**tement ordered **T**able of **E**xecution **C**onditions), between each statement (in the normalized program) and the execution conditions of its non-affine dimensions. These are the execution conditions of non-affine for loops and if statements. Additionally we remember the origin of each statement:

types of origins	
ordinary	the statement was an ordinary statement in the source program
if	the statement was an if statement that was normalized to a *new_if* array
$UB[]$	the statement is the new assignment of the original non-affine upper bound of a non-affine for loop
$maxfubb$	the statement is the new assignment which calculates the maximum of the upper bounds of a non-affine for loop
while	the statement is a *new_if* statement originating from the normalization of a while loop

Note that we do not have to store the % expressions concerning the strides and the conditions of affine for loops, as these are already considered by the code generator for affine loop nests.
We use relation STATEC to retransform the target program (see Section 4.2).

Example 13 (STATEC).
This example is a continuation of Example 12 and shows relation STATEC for the normalized source program.

STATEC for the normalized program		
statement	conditions	origin
7	—	if
4	$new\_if_1[i] = tt$	ordinary
5	$new\_if_1[i] = tt$	$UB[]$
6	$new\_if_1[i] = tt$	$maxfubb$
1	$new\_if_1[i] = tt$ $0 \leq j \leq UB[i]$	ordinary
2	$new\_if_1[i] = tt$ $0 \leq j \leq UB[i]$	ordinary
8	—	while
3	$new\_if_2[i,k] = tt$	ordinary

Each row expresses the relation between a statement and the conditions that determine the dimensions of its execution space.

After the normalization, the source program meets the requirements we have imposed on the retransformation, and we can send it through the "parallelization pipeline" of LooPo.

4.2 Retransformation

We base our method on the "synchronous run time" target program produced by the method described in [25]. This computes already the target loop bounds and *executed* predicates for the normalized (and therefore affine portions of the) execution spaces. So we only have to restore the changes made during normalization.

Remark. The retransformation part could also be implemented as an integral component of the target generator by applying the extensions described in the previous chapter.

See Example 14 for an accompanying illustration of the retransformation steps.

Input:
- the (partitioned, tiled) target code generated by the "synchronous run time method" of the target code generator by Wetzel [25],
- a list of records that establish a relation between every statement and its (non-affine) dimensions.

Output: the retransformed synchronous target program with the same semantics as the source program

1. *Retransformation of the Target Loops*:

   Scan the target program for target loops. Sequential loops (for loops) are time loops and parallel loops (parfor loops) are space loops.

   (a) *Time Loops*:

   A time loop looks as follows:

   for $t := \min(tlb_1, \ldots, tlb_n)$ to $\max(tub_1, \ldots, tub_n)$ step ST do
   $\quad\ldots$
   end

   t is the index of the time loop, n is the number of program parts in the target program generated by the target generator of Wetzel. tlb_k (tub_k), $1 \leq k \leq n$, are the lower (upper) bounds of the respective dimension of the scanned target space for program part k. The tlb_k and tub_k are ordered according to the sequence of the program parts in the target program and each pair (tlb_k,tub_k) corresponds to exactly one program part.

   In the last chapter, we have shown that a lower bound of a time dimension never depends on an upper bound of a non-affine loop. Thus, no expression tlb_k can contain a *FUBB* or *WUBB*.

   Consequently we only have to change the upper bounds of time loops. Let tub_k be an upper bound that contains some *WUBB*s and *FUBB*s, denoted

   $$tub_k(FUBB_1, \ldots, FUBB_l,\ WUBB_{l+1}, \ldots,\ WUBB_m)$$

   We have to replace the *FUBB*s and *WUBB*s by the respective variables *maxfubb* and *maxwubb*, introduced during normalization and computed at run time. This means that the upper bound changes during the execution of the target program and can be taken into account by changing the for loop into a while loop.

   The values of *maxfubb*, however, are not valid from the beginning of the execution, so we must prevent them from being read "too early" for the evaluation of the upper bound of time.

If some invalid *maxfubb*s occur in a tub_k expression, this means that no operation of the corresponding statement has to be executed yet (ensured by the dependences pointing from the *maxfubb* statements to all the body statements). The tub_k should appear as if it was not used for calculating the upper bound of the time loop. We can achieve this by using the upper bound, say tub_o, of any other program part which is sure not to contain any *FUBB*s. The upper bound of the respective time dimension of the *maxfubb* statement belonging to the outermost non-affine for loop in the loop nest of statement k meets this requirement since it is not in the body of any non-affine for loop.

To decide whether some *maxfubb*s are valid or not, we utilize the respective *validfubb*s in combination with the '$cond?a : b$' operator. It corresponds to the conditional operator known from the programming language C and evaluates to 'a' if '$cond$' is *tt* and to 'b' if '$cond$' is *ff*.

The substitute for tub_k is:

$$(validfubb_1 \text{ and } \ldots \text{ and } validfubb_l\ ?\ tub_k(maxfubb_1, \ldots, maxfubb_l, maxwubb_{l+1}, \ldots, maxwubb_m) : tub_o)$$

If k is the only statement whose upper bound of the time dimension depends on *FUBB*s and *WUBB*s, the whole time loop looks like this:

```
for t := min(tlb_1, ..., tlb_n)
        while t <= max(tub_1, ..., tub_{k-1},
                       validfubb_1 and ... and validfubb_l ?
                       tub_k(maxfubb_1    , ..., maxfubb_l,
                             maxwubb_{l+1}, ..., maxwubb_m) :
                       tub_o,
                       tub_{k+1}, ..., tub_n) step ST
do
   ...
end
```

(b) *Space Loops*:

A space loop looks as follows:

```
parfor p := min(tlb_1, ..., tlb_n) to max(tub_1, ..., tub_n) step ST do
   ...
end
```

p is the index of the space loop, n is again the number of program parts in the target program. tlb_k (tub_k), $1 \leq k \leq n$, are the lower (upper) bounds of the respective dimension of the scanned target space for statement k.

The upper bounds of space loops must be handled the same way as the upper bounds of time loops. However, space loops (according to the proof of Lemma 43 in the previous chapter) do not have to be changed to while loops.

In contrast to time loops, also the lower bounds of space loops may depend on *FUBB*s and *WUBB*s, so these must be considered, too. Let tub_k be such a lower bound, denoted by

$$tlb_k(FUBB_1, \ldots, FUBB_l, WUBB_{l+1}, \ldots, WUBB_m)$$

Again the *WUBB*s are changed to the respective *maxwubb*s.

Instead of using the tub_o if the occurring *validfubb*s are *ff*, we use the respective tlb_o for lower bounds of space loops.

The substitute for a tlb_k is:

$$(\mathit{validfubb}_1 \text{ and } \ldots \text{ and } \mathit{validfubb}_l\ ? \\ tub_k(\mathit{maxfubb}_1, \ldots, \mathit{maxfubb}_l, \mathit{maxwubb}_{l+1}, \ldots, \mathit{maxwubb}_m) : tlb_o)$$

The explained steps must be repeated for every element in the min and max expressions of the target loop bounds and for every target loop.

2. *Retransformation of the executed Predicates*:

   Scan the target program statement by statement. Each statement is guarded by an if statement of the form:

   $$\begin{array}{l} \text{if } \mathit{scond}_1 \text{ and } \ldots \text{ and } \mathit{scond}_p \text{ then} \\ \quad s' \\ \text{endif} \end{array}$$

   s' is the transformed statement s and the $scond_q$, $1 \leq q \leq p$, are subconditions depending on the values of the target indices. They do not contain or/and operators.

   If the source program contained only affine loops, these if statements would describe the execution spaces of the statements in their bodies.

However, since we changed the original loop bounds, we do not get the original execution spaces now. In this step, we correct these *executed* predicates.

For every statement, we have to look at the subconditions of the surrounding if statement and at the source execution spaces. Accordingly we distinguish several cases:

(a) *The Statement Belongs to the Bodies of* if *Statements in the Source Program.*

During normalization we introduce a *new_if* array that stores the values of the condition of each if operation. Since the execution spaces of the statements in the body of if statements depend on these values, we have to guard the current statement by another if statement whose condition checks whether the surrounding transformed *new_if* arrays all contain the value *tt*.

(b) *No Subcondition Contains a FUBB or WUBB.*

This implies that the target execution space of the corresponding statement does not depend on the bounds of non-affine loops. Nothing has to be done.

(c) *A Subcondition Contains FUBBs.*

This means that the execution space of the corresponding statement depends on the upper bound of a non-affine for loop. Let $scond_q(FUBB_1, \ldots, FUBB_l)$ be such a subcondition.

The *FUBB*s must be replaced by the real values of the upper bounds of the respective loops computed by the *UB* statements that were introduced during normalization. The source indices in the index lists of *UB* arrays must be replaced by their image under the transformation function for the current target statement. These can be computed from the inverse transformation.

The dependences from the *UB* statements to all body statements ensure that the *UB* arrays are always computed before they are needed. However, since the target execution spaces cannot be scanned exactly, the target program references array elements that are not written yet or do not exist at all.

To prevent accesses to undefined array elements, we guard the whole program part by a new if condition that checks whether the respective values of all *valid_UB* arrays (see also Item 3a) yield *tt*. If so, the referenced values of the *UB* arrays may be read. Otherwise, the respective elements of the *UB* arrays were not written before and the whole program part must not be executed. The substitute for the current program part looks like this:

```
if valid_UB_1 and ... and valid_UB_l then
   if scond_1 and ... and
      scond_q(UB_1, ..., UB_l) and ... and scond_p
   then
      s'
   endif
endif
```

The UB_k are the new arrays that were introduced during the same normalization step as the corresponding $FUBB_k$.

(d) *A Subcondition Contains WUBBs*

In this case the current target statement belongs to the body of a while loop. Let $scond_q(WUBB_1, \ldots, WUBB_l)$ be such a subcondition.

Like in the target loop bounds, we replace each $WUBB_k$, $1 \leq k \leq l$, by its corresponding $maxwubb_k$ This yields the whole scanned target space for this statement.

If the statement is a *new_if* statement originating from a while loop, the actual execution space does not matter, i.e., the statement may be executed for every scanned iteration (see the additional descriptions for *new_if* statements originating from a while loop in Item 3c).

Is the current statement an 'ordinary' statement (not originating from a while loop), then we have to guard it by the value of the respective *new_if* statement, just as if it were in the body of a regular if statement.

3. *Retransformation of the Program Part Bodies*:

In this step we insert the code for the administration of the new variables *UB*[...], *validfubb*, *maxfubb* and *maxwubb*.

(a) *The Current Statement is a UB Statement that Computes the Upper Bound of a Non-Affine* for *Loop*

To indicate the elements of the *UB* arrays that contain valid values, we introduce an additional boolean array *valid_UB* for each

UB array. It is indexed with the same indices and an element is set to *tt* if the corresponding element in the *UB* array contains a valid value and to *ff* in all other cases. The new program part looks like this:

$$\begin{array}{ll} \textsf{if} \ \ldots \ \textsf{then} & // \text{ as in the original part} \\ \quad UB[indlist] := \ldots & // \text{ as in the original part} \\ \quad valid\_UB[indlist] := tt & // \text{ new assignment} \\ \textsf{endif} & \end{array}$$

We describe how we handle accesses to the new boolean arrays (*valid_UB* and *new_if*) in Section 4.3.

(b) *The Current Statement is a maxfubb Statement Originating from a Non-Affine* for *Loop*:

These statements must take care of the administration of the *validfubb* variables. Thus, if the corresponding *maxfubb* is written for the first time, the *validfubb* must be set to *tt*. Since not every processor needs to do this, we choose to select the one with the minimal space coordinates.

The following piece of code is inserted behind the current statement:

$$\begin{array}{l} \textsf{if} \ (validfubb = ff \ \textsf{and} \ p_1 = tlb_{p_1} \ \textsf{and} \ \ldots \ \textsf{and} \ p_{d^{\mathrm{p}}} = tlb_{p_{d^{\mathrm{p}}}}) \ \textsf{then} \\ \quad validfubb := tt \\ \textsf{endif} \end{array}$$

d^{p} is the number of space loops. The indices of the space loops are the ps, and the tlbs are the lower bounds of the respective space dimensions for the current statement.

(c) *The Current Statement is a new_if Statement Originating from a* while *Loop*:

These statements serve to calculate the execution conditions of the respective while loops by following Definition 29 in Chapter 2 on page 26.

The only difference is that the source indices are expressed by the respective (affine) combination of the target indices, according to the inverse transformation (see also Example 14).

If the execution condition yields *tt*, the value of the respective *maxwubb* must be updated: $maxwubb = \max(maxwubb, i + 1)$ where i is the index of the respective while loop (see also Figure 3.3).

We do not need any barrier statements as described in [1], because we do not calculate an explicit *terminated* predicate. The synchronous model ensures that all values are changed before the next time step is initiated.
The following example illustrates the various retransformation steps.

Example 14 (Retransformation).
This example is a continuation of Example 12. The Darte-Vivien scheduler [2, 21] yields the following schedules and allocations:

statement	transformation	inverse transformation
7	$t_0 = 0$ $p_1 = i$ $p_2 = 0$	$i = p_1$
4 5	$t_0 = 1$ $p_1 = i$ $p_2 = 0$	$i = p_1$
6	$t_0 = 2$ $p_1 = i$ $p_2 = 0$	$i = p_1$
1 2	$t_0 = 3$ $p_1 = i$ $p_2 = j$	$i = p_1$ $j = p_2$
8	$t_0 = 2 * k$ $p_1 = i$ $p_2 = 0$	$i = p_1$ $k = t_0/2$
3	$t_0 = 2 * k + 1$ $p_1 = i$ $p_2 = 0$	$i = p_1$ $k = (t_0 - 1)/2$

We illustrate the retransformation by means of the "synchronous run time" output of our target generator [25] for the given schedules and allocations.

```
/* synchronous run time output */

for t0 := ceil(min(0,1,1,2,3,3,0,1)) to
          floor(max(0,1,1,2,3,3,2 * WUBB,2 * WUBB + 1)) do
  parfor p1 := ceil(min(0,0,0,0,0,0,0,0)) to
               floor(max(4,4,4,4,4,4,4,4)) do
    parfor p2 := ceil(min(0,0,0,0,0,0,0,0)) to
                 floor(max(0,0,0,0,FUBB,FUBB,0,0)) do
```

if ($0 <= t_0$ and $t_0 <= 0$ and $t_0\%1 = 0$ and
$0 <= p_1$ and $p_1 <= 4$ and $p_1\%1 = 0$ and
$0 <= p_2$ and $p_2 <= 0$ and $p_2\%1 = 0)$
then
7 $new_i f_1[p_1] := (a[p_1] = 0)$
endif

if ($1 <= t_0$ and $t_0 <= 1$ and $(t_0 - 1)\%1 = 0$ and
$0 <= p_1$ and $p_1 <= 4$ and $p_1\%1 = 0$ and
$0 <= p_2$ and $p_2 <= 0$ and $p_2\%1 = 0)$
then
4 $LB[p_1] := b[p_1]$
endif

if ($1 <= t_0$ and $t_0 <= 1$ and $(t_0 - 1)\%1 = 0$ and
$0 <= p_1$ and $p_1 <= 4$ and $p_1\%1 = 0$ and
$0 <= p_2$ and $p_2 <= 0$ and $p_2\%1 = 0)$
then
5 $UB[p_1] := \mathrm{abs}(c[p_1])$
endif

if ($2 <= t_0$ and $t_0 <= 2$ and $(t_0 - 2)\%1 = 0$ and
$0 <= p_1$ and $p_1 <= 4$ and $p_1\%1 = 0$ and
$0 <= p_2$ and $p_2 <= 0$ and $p_2\%1 = 0)$
then
6 $maxfubb := max(maxfubb, UB[p_1])$
endif

if ($3 <= t_0$ and $t_0 <= 3$ and $(t_0 - 3)\%1 = 0$ and
$0 <= p_1$ and $p_1 <= 4$ and $p_1\%1 = 0$ and
$0 <= p_2$ and $p_2 <= FUBB$ and $p_2\%1 = 0)$
then
1 $b[p_2 + LB[p_1]] := b[p_2 + LB[p_1]] + 1$
endif

if ($3 <= t_0$ and $t_0 <= 3$ and $(t_0 - 3)\%1 = 0$ and
$0 <= p_1$ and $p_1 <= 4$ and $p_1\%1 = 0$ and
$0 <= p_2$ and $p_2 <= FUBB$ and $p_2\%1 = 0)$
then
2 $c[p_2 + LB[p_1]] := c[p_2 + LB[p_1]] + 1$
endif

```
              if (  0 <= t0  and  t0 <= 2 * WUBB  and  t0%2 = 0  and
                    0 <= p1  and  p1 <= 4         and  p1%1 = 0  and
                    0 <= p2  and  p2 <= 0         and  p2%1 = 0)
              then
8                 new_if2[2 * p1/2, t0/2] := (a[2 * p1/2] > 0)
              endif

              if (  1 <= t0  and  t0 <= (2 * WUBB + 1)  and  (t0 - 1)%2 = 0  and
                    0 <= p1  and  p1 <= 4               and        p1%1 = 0  and
                    0 <= p2  and  p2 <= 0               and        p2%1 = 0)
              then
3                 a[2 * p1/2] := a[2 * p1/2] - 1
              endif

            end
        end
    end
```

The functions ceil, floor, abs, min and max are predefined and compute the next greater integer, the next smaller integer and the absolute value of a number and the minimum and maximum of a tuple of numbers, respectively.

The first target loop is a time loop whose upper bound depends on the upper bound of a while loop in the source program, indicated by the $WUBB$ in its upper bound:

```
for t0 :=  ceil(min(0, 1, 1, 2, 3, 3, 0, 1)) to
           floor(max(0, 1, 1, 2, 3, 3, 2 * WUBB, 2 * WUBB + 1)) do
   ...
end
```

Accordingly we must change this loop to a while loop where $WUBB$ is replaced with the new variable $maxwubb$:

```
for t0 := ceil(min(0, 1, 1, 2, 3, 3, 0, 1))
        while t0 <= floor(max(0, 1, 1, 2, 3, 3,
                              2 * maxwubb, 2 * maxwubb + 1)) do
   ...
end
```

The second target loop is a space loop. No bound depends on a non-affine loop in the source program, so nothing has to be done here.

The third target loop is again a space loop, but this time the upper bound depends on the upper bound of the non-affine for loop in the source program. We have to replace *FUBB* by *maxfubb* and guard it by the '?:' operator.
The statements corresponding to the elements of the max expression that depend on the *FUBB* are statement 1 and 2. The outermost non-affine for loop is the loop with index j and the respective *maxfubb* statement is statement 6. As statement 6 corresponds to the fourth program part in the target program, we set the second branch of the '?:' operator to the fourth element in the max expression, i.e., to 0:

```
parfor p2 := ceil(min(0, 0, 0, 0, 0, 0, 0, 0)) to
              floor(max(0, 0, 0, 0, validfubb ? maxfubb : 0,
                                    validfubb ? maxfubb : 0, 0, 0)) do
    ...
end
```

We have retransformed the target loops to enumerate the target space described in the last chapter. The execution conditions must still be retransformed. This is done statement by statement.

The execution space of statement 7 only depends on the range of an affine for loop, so nothing needs to be done here.
The execution spaces of statements 4, 5 and 6 depend additionally on the value of the if condition evaluated by statement 7. These statements may only be executed if the respective evaluation of the if condition yielded *tt*. So we have to check this value. It is stored in $new\_if_1[i]$, where i must be recomputed from the inverse of the transformation function.
We can gather from the table at the beginning of Example 14 that the value of i equals the value of p_1, so we must replace i by p_1 in $new\_if_1[i]$ and get $new\_if_1[p_1]$. The new program part for statements 4 and 5 is:

if $(new\_if_1[p_1] = tt)$ then
  if ($1 <= t_0$ and $t_0 <= 1$ and $(t_0 - 1)\%1 = 0$ and
    $0 <= p_1$ and $p_1 <= 4$ and $p_1\%1 = 0$ and
    $0 <= p_2$ and $p_2 <= 0$ and $p_2\%1 = 0$)
  then
4    $LB[p_1] := b[p_1]$
  endif
endif

if $(new\_if_1[p_1] = tt)$ then
  if ($1 <= t_0$ and $t_0 <= 1$ and $(t_0 - 1)\%1 = 0$ and
    $0 <= p_1$ and $p_1 <= 4$ and $p_1\%1 = 0$ and
    $0 <= p_2$ and $p_2 <= 0$ and $p_2\%1 = 0$)
  then
5    $UB[p_1] := \mathrm{abs}(c[p_1])$
    $valid\_UB[p_1] := tt$
  endif
endif

As statement 5 evaluates the UB array for the upper bound of the non-affine for loop, we added the additional assignment $valid\_UB[p_1] := tt$ to indicate that the value of $UB[p_1]$ was written.
Statement 6 is the statement that calculates the maximum of the upper bounds of the non-affine for loop. It also serves to indicate that $maxfubb$ is valid after it received its first value:

if $(new\_if_1[p_1] = tt)$ then
  if ($2 <= t_0$ and $t_0 <= 2$ and $(t_0 - 2)\%1 = 0$ and
    $0 <= p_1$ and $p_1 <= 4$ and $p_1\%1 = 0$ and
    $0 <= p_2$ and $p_2 <= 0$ and $p_2\%1 = 0$)
  then
6    $maxfubb := max(maxfubb, UB[p_1])$
    if $validfubb = ff$ and $p_1 = 0$ and $p_2 = 0$ then
      $validfubb = tt$
    endif
  endif
endif

The execution spaces of statements 1 and 2 depend (besides the i-loop) on the range of the non-affine for loop with index j and on the value of the if condition $new\_if[i]$.
The dependence on the if condition is handled like for statement 4. The execution conditions computed by the target code generator contain a $FUBB$ because the statements belong to the body of a non-affine for loop.
We have to exchange the $FUBB$ by the variable for the real upper bound of the respective instance of the j-loop. This must be done for both statements $UB[p_1]$, as $i = p_1$ (see the table at the beginning of Example 14). To prevent uninitialized elements of $UB[p_1]$ from being accessed, we check the respective value of $valid\_UB[p_1]$ (second if statement).
The transformed program parts for statements 1 and 2 appear as follows:

```
    if (new_if_1[p1] = tt) then
      if (valid_UB[p1] = tt) then
        if ( 3 <= t0  and  t0 <= 3       and  (t0 - 3)%1 = 0  and
             0 <= p1  and  p1 <= 4       and        p1%1 = 0  and
             0 <= p2  and  p2 <= UB[p1]  and        p2%1 = 0)
        then
1         b[p2 + LB[p1]] := b[p2 + LB[p1]] + 1
        endif
      endif
    endif

    if (new_if_1[p1] = tt) then
      if (valid_UB[p1] = tt) then
        if ( 3 <= t0  and  t0 <= 3       and  (t0 - 3)%1 = 0  and
             0 <= p1  and  p1 <= 4       and        p1%1 = 0  and
             0 <= p2  and  p2 <= UB[p1]  and        p2%1 = 0)
        then
2         c[p2 + LB[p1]] := c[p2 + LB[p1]] + 1
        endif
      endif
    endif
```

The next statement is statement 8. Its execution space depends on the i-loop and the while loop with new index k. It was not really an if statement, but originates from the while loop and has to compute the execution condition for the other statements in the body.
The upper bound of the target index t_0 depends on the artificial constant $WUBB$. We must replace it by the proper value $maxwubb$.

Then the if statement around statement 8 expresses its scanned target space. To get the target execution space, we have to implement the rest of the execution condition. With $k = t_0/2$ from the table at the beginning of Example 14, we get the following part for statement 8:

if ($0 <= t_0$ and $t_0 <= 2 * maxwubb$ and $t_0 \% 2 = 0$ and
    $0 <= p_1$ and $p_1 <= 4$ and $p_1 \% 1 = 0$ and
    $0 <= p_2$ and $p_2 <= 0$ and $p_2 \% 1 = 0$)
then
  if ($t_0/2 = 0$ and $a[2 * p_1/2] > 0$)
  then
8_1 $new\_if_2[2 * p_1/2, t_0/2] := tt$
    $maxwubb = \max(maxwubb, t_0/2 + 1)$
    if ($validwubb = ff$) then $validwubb := tt$ endif
  else
    if ($t_0/2 > 0$ and $new\_if_2[2 * p_1/2, (t_0/2) - 1] = tt$ and
     $a[2 * p_1/2] > 0$)
    then
8_2 $new\_if_2[2 * p_1/2, t_0/2] := tt$
      $maxwubb = \max(maxwubb, t_0/2 + 1)$
    else
8_3 $new\_if_2[2 * p_1/2, t_0/2] := ff$
    endif
  endif
endif

We still have statement 3 to consider. It is an ordinary assignment whose execution space depends on the affine for loop and the while loop. We have to change the upper bound of t_0 in the if statement as we did for statement 8. This again yields the scanned target space. To get the target execution space, we must check the value of the execution condition of the while loop. It is stored in $new\_if_2[i, k]$. For this statement, $i = p_1$ and $k = (t_0 - 1)/2$. We get the result:

if ($new\_if_2[p_1, (t_0 - 1)/2] = tt$) then
  if ($1 <= t_0$ and $t_0 <= 2 * maxwubb + 1$ and $(t_0 - 1) \% 2 = 0$ and
     $0 <= p_1$ and $p_1 <= 4$ and $p_1 \% 1 = 0$ and
     $0 <= p_2$ and $p_2 <= 0$ and $p_2 \% 1 = 0$)
  then
3     $a[2 * p_1/2] := a[2 * p_1/2] - 1$
  endif
endif

If we put all these program parts together we obtain the retransformed target program that executes only the operations that belong to the target execution space of a statement.

4.3 The Size of the New Arrays

During normalization and retransformation we have introduced various new arrays for storing lower bounds, upper bounds and the values of conditions. However, we did not consider the size of these arrays so far.
We index them with the indices that may belong to dynamic loops. In this case — as for the execution spaces — we cannot give a precise size at compile time. For non-affine dimensions we must assume an initial range. If it turns out to be too small at run time, it is necessary to resize the array. This is an expensive operation and it shows the omnipresent tradeoff between run time and memory consumption: if the ranges of non-affine dimensions are by chance estimated large enough no resizing is necessary and no run time overhead is caused.
We want to present a possible high-level way to handle the accesses to such *dynamic arrays*, but leave the concrete implementation up to the target output module that maps the internal representation to a real programming language.

4.3.1 Dynamic boolean Arrays

A new dynamic boolean array is used to indicate either which elements of a corresponding integer array may be accessed ($valid\_UB[]$) or which instances of an execution condition yield *tt* or *ff* ($new\_if[]$).
We propose the following data structure for controlling the accesses to these dynamic arrays.

structure dynamic_boolean_array
begin
  integer $lb_1 := DEFAULT\_LB_1,$
  $\dots$
  $lb_d := DEFAULT\_LB_d,$
  $ub_1 := DEFAULT\_UB_1,$
  $\dots$
  $ub_d := DEFAULT\_UB_d,$
  boolean $dyn\_array[ub_1 - lb_1] \dots [ub_d - lb_d]$
end

The structure stores the current upper bounds of the contained array *dyn_array*. This enables the functions $get()$ and $set()$ to check whether the referenced element is inside the current range of the arrays bounds. If so, and if the $value$ passed to $set()$ equals *tt*, the referenced element is set equal to $value$.

Remark. By convention all elements are considered to be initialized to *ff* (even the elements that are currently not in the range of the array). So *ff* values do actually not have to be written.

If the value *tt* should be written into an element that does not exist yet, the $resize()$ function must be called. It has to allocate a boolean array that is large enough to include the new element. The values of the old elements must be copied into the new array. The additional elements must be set to *ff*. After that, the old array can be deleted and replaced by the new array. The respective values of the bounds, $lb_1, \dots, lb_d$ and $ub_1, \dots, ub_d$ must be adjusted to the new bounds.

```
function set(dynamic_boolean_array array, boolean value,
             integer 1st, ..., integer dth)
begin
   if (value = ff) then
      return
   endif
   if (array.lb_1 <= 1st - (array.lb_1) <= array.ub_1) and ... and
      (array.lb_d <= dth - (array.lb_d) <= array.ub_d)
   then
      array.dyn_array[1st - (array.lb_1), ..., dth - (array.lb_d)] := value
   else
      resize(array, 1st, ..., dth)
      array.dyn_array[1st - (array.lb_1), ..., dth - (array.lb_d)] := value
   endif
end
```

Function $get()$ returns the value of the specified array element if it is inside the current extent of the array, else *ff* is returned.

```
function get(dynamic_boolean_array array,
             integer 1st, ..., integer dth) : boolean
begin
   if (array.lb_1 <= 1st - (array.lb_1) <= array.ub_1) and ... and
      (array.lb_d <= dth - (array.lb_d) <= array.ub_d)
   then
      return(array.dyn_array[1st - (array.lb_1), ..., dth - (array.lb_d)])
   else
      return(ff)
   endif
end
```

The construction we presented ensures that arbitrary elements of our dynamic boolean arrays may be referenced and the program reacts with a defined behavior, even if a referenced element does not exist.

4.3.2 Dynamic integer Arrays

The second type of dynamic array we introduce during normalization stores the values of the lower and upper bounds of non-affine for loops ($LB[]$ and

$UB[]$). This might also be indexed with the loop indices whose range depends on non-affine dimensions and therefore its size cannot be predicted at compile time.
For dynamic integer arrays the definition of the structure changes slightly, namely the type of the *dyn_array* elements changes to integer:

$$
\begin{array}{l}
\textsf{structure dynamic\_integer\_array} \\
\textsf{begin} \\
\quad \begin{array}{lrcl}
\textsf{integer} & lb_1 & := & DEFAULT\_LB_1, \\
 & \ldots & & \\
 & lb_d & := & DEFAULT\_LB_d, \\
 & ub_1 & := & DEFAULT\_UB_1, \\
 & \ldots & & \\
 & ub_d & := & DEFAULT\_UB_d, \\
\textsf{integer} & \multicolumn{3}{l}{dyn\_array[ub_1 - lb_1]\ldots[ub_d - lb_d]}
\end{array} \\
\textsf{end}
\end{array}
$$

In contrast to dynamic boolean arrays, the $set()$ function for dynamic integer arrays must write each value passed to it.

$$
\begin{array}{l}
\textsf{function } set(\textsf{dynamic\_integer\_array } array, \textsf{integer } value, \\
\qquad\qquad \textsf{integer } 1st, \ldots, \textsf{integer } dth) \\
\textsf{begin} \\
\quad \textsf{if } (array.lb_1 <= 1st - (array.lb_1) <= array.ub_1) \textsf{ and } \ldots \textsf{ and} \\
\qquad (array.lb_d <= dth - (array.lb_d) <= array.ub_d) \\
\quad \textsf{then} \\
\qquad array.dyn\_array[1st - (array.lb_1), \ldots, dth - (array.lb_d)] := value \\
\quad \textsf{else} \\
\qquad resize(array, 1st, \ldots, dth) \\
\qquad array.dyn\_array[1st - (array.lb_1), \ldots, dth - (array.lb_d)] := value \\
\quad \textsf{endif} \\
\textsf{end}
\end{array}
$$

For this kind of array, we ensure (by using the surrounding if statements) that only previously written elements are read: every read value is a valid value, unread elements may contain undefined values. This means that the $get()$ function does not have to check the current range of the referenced dynamic integer array.

function get(dynamic_integer_array $array$,
integer $1st, \ldots,$ integer dth) : integer
begin
return($array.dyn\_array[1st - (array.lb_1), \ldots, dth - (array.lb_d)]$)
end

The resize function should extend the size of the arrays in as large as possible steps to keep the probability of write accesses to not existing elements as small as possible.

We have described all steps and elements of the implementation of our method for estimating dynamic execution spaces at run time now. The last chapter gives an overview of the thesis, some prospects on possible future work and some general thoughts.

Chapter 5

Conclusion

Let us summarize the major concepts and their correlation in the setting of parallelization in the polyhedron model. The main task in our thesis was to describe (dynamic) target execution spaces by loop nests at compile time. Three problems arose:

1. In general, loop nests are too weak to describe these spaces exactly (scannability).

2. There are types of execution spaces whose shape cannot be predicted at compile time.

3. There are types of execution spaces whose shape *and extent* cannot be predicted at compile time.

Point 1 implies that we must enumerate iterations at which no operations have to be executed, and this leads to the distinction between *index* and *execution space*. Since we scan too much, we must find a way to filter the iterations at which some operation has to be executed. This is done by the execution conditions and the *executed* predicate described in Chapter 3.
Point 2 alludes to the kind of static execution space that may be known entirely at compile time, however, the shape of its corresponding target execution space cannot be described by loop bounds — due to technical or theoretical reasons. For this kind of execution space, there is an index space whose shape can be described by target loop bounds and which contains the actual execution space as a subset. *Execution determination* must be applied but no termination detection is necessary.
Finally there are the *dynamic execution spaces* (Point 3) whose shape and extent cannot be predicted at compile time, but must be estimated at run

time. Since the approximation may not be precise, we must care for *execution determination* and have to establish the extent via *termination detection*.
The processing of dynamic execution spaces is the main topic of this thesis and one method is described in Chapter 3.

We can imagine some extensions to our classification (in Chapter 2). One case we do not consider is the kind of execution space that needs termination detection but no execution determination. Consider the following loop nest:

```
while (cond) do
   for i := 0 to 10 do
      s;
   end
end
```

Although there is a dynamic dimension, we know the shape of the execution space — it is a (rectangular) polytope. However, we do not know the extent of the first dimension. We know that polytopes belong to the class of affine execution spaces and can be scanned precisely, so we only have to apply termination detection.
Without further examination, we believe that only this kind of loop nest (the outermost loop is a while loop, all inner loops are affine for loops) contains execution spaces that need termination detection but not execution determination.

Another conceivable extension to the classification is a class that contains a cross between static and dynamic execution spaces; we call them semi-dynamic execution spaces.
Unlike static execution spaces, there are dependences pointing to the bound expressions, but the sources of these dependences are in the execution space just scanned, i.e., they are not caused by a statement that belongs to the body of the current loop nest. Thus, although the shape and extent of the current execution spaces are not known at compile time, they are known before the execution of the loop nest starts. The assessment of possible positive implications on target code generation is left for future work.

Let us recall that the classification in Chapter 2 is only done with respect to the execution space of one single statement. These separate spaces must be merged to one complete target space that contains at least the union of all separate target execution spaces; this means scanning on a different level. Although each separate execution space is scannable the union may not be. The effects are the same: non-scannability requires execution determination.

The scannability of the separate target execution spaces of each statement is only a necessary condition, but if we want to exploit the benefit of scannability the union of all execution spaces must be scannable, too. This is not taken into account in the classification and that is why (separate) scannable transformations do not yield as much benefit as they seem to promise at first sight: they ensure easier termination detection but still need execution determination because of the necessity of merging.

We want to make another remark on merging here. In our thesis merging is viewed as finding a *perfect* target loop nest that enumerates the union of execution spaces. However, we can imagine merging methods that yield imperfectly nested target loops. This would mean that (parts of) an unscannable target execution space can be divided into scannable subsets. Thus, some parts of the execution determination are already incorporated in the target loop bounds and need not be evaluated as separate execution predicates. Maybe this would allow for better run time results.

We implement termination detection as side effect of execution determination. However, the correlation is not necessarily as close as this suggests. The implication is: if there are no operations executed now and in the future then the target program can be terminated.
If someone gives us a better method for termination detection (maybe hardware-supported) then we can remove the code for managing the *validfubb*s, *maxfubb*s and *maxwubb*s and change the target loops according to the new method.
Another option would be to iterate time for a given number of steps (say n) and then check if the program should already have terminated. The disadvantage of this method is that the program runs longer than necessary. On the other hand, it may be possible that the overhead for termination detection can be decreased to $1/n$-th (compare also to a clever resizing of the dynamic arrays). Enumerating too much time does not affect the correctness of the results, as execution determination compensates for every inaccuracy of scanning.

Let us point out some aspects of our implementation and possible modifications. In Chapter 3 we explained that we do not need a `while` dependence for non-affine `for` loops, because the sequence of computing the *maxfubb*s is arbitrary. This realization allows for 'good' ways to compute maxima, e.g., parallel algorithms that need only logarithmic time. However, our target output module will not be able to use this facility yet, so we chose to allow the dependence analysis to find the data dependence from one instance of the *maxfubb* statements to the next instance.

By nature, affine loop bounds do not cause side effects. Our normalization method of storing the non-affine loop bounds and the values of if conditions in arrays allows for loop bounds and if conditions with side effects.
We set the lower bound of a while loop always to 0. It may turn out during future studies that it might be useful to have different lower bounds, e.g., to reduce communication. Our theory allows for all kinds of affine lower bounds for while loops, just as for for loops.
It is possible to permit the usage of indexed while loops already in the source program. In this case the lower bound must be affine and the index must be unique. Combined with a generalization of Section 4.3, the user may write source programs that contain dynamic arrays — always with the 'danger' of a possibly necessary resizing.
The dynamic arrays can be spread across the processors such that the resize operations are local to one processor: not the whole array is resized but only the part of the array that is located on one processor. This saves time and memory and fits the view of distributed memory.
The *valid_UB* arrays that indicate whether an element of a dynamic integer array does exist and is initialized is not needed necessarily. Instead the respective value could be computed each time it is referenced: if the *executed* predicate of the statement that should have written the referenced element is *tt* then the value is valid and can be read. There is (as so often) a great variety of design decisions until a concrete realization of the theory will be reached. Different implementations and performance measurements will have to show the practical characteristics of the various implementations.

For each iteration of a while loop we remember whether it is executed or not. For non-affine for loops we store the range of the index instead. This means reduced memory overhead, as the storage of the values of the *executed* predicates is pulled to the next outer level.
On the other hand: the indices of while loops show the same behavior as the indices of for loops, i.e., they also cover a range between a lower and an upper bound. The only difference is that the upper bound must be computed at run time. Thus, we can use the same storage scheme as for non-affine for loops. The source program

```
for i := 1 to N do
    while cond do
        body;
    end
end
```

could be normalized to

```
for i := 1 to N do
   UB[i] := cond?0 : -1;
   for j := 0 to WUBB step ST do
      body;
      UB[i] = cond?UB[i] + ST : UB[i];
      maxwubb = max(maxwubb, UB[i]);
   end
end
```

In the target loop bounds $maxwubb$ is used as before, but now in the *executed* predicates we use the new $UB[i]$ array just as we do for non-affine for loops. We must also add the appropriate control dependences from the UB and $maxwubb$ statements to the *body*.
Note that UB is only indexed with i and not with i and j as the *new_if* array we introduce during normalization.

Remark. The only advantage of this method is the memory reduction by one dimension. The computation of the bounds is still placed at the same level as the while loop.

At the very end of this thesis let us make some general — partly philosophical — remarks on the topic of parallelization which we have learned to keep in mind when we are thinking about parallelism:

- The fact that an execution space is bounded does not imply that the bounds are known. Since the information about the execution space changes during run time, it is in general impossible to enumerate the complete execution space in one time step. Instead, different parts of it must be enumerated successively, i.e., the task of enumerating a space itself costs time.

- In general, it is impossible to evaluate the quality of the whole parallelization by only looking at the results of one parallelization step.

  E.g., finding descriptions of the target loop bounds at compile time is inherent to the polyhedron model. If this seems to destroy possible parallelism indicated by the transformation of a statement, the reason is not necessarily a bad implementation of the target generator, but may also be the restricted view of the schedulers and allocators (which is sufficient for them) on the parallelization problem.

Of course we have to look for ways to compute the execution spaces as efficiently as possible and it should be possible to find theoretical bounds for the degree of this efficiency. These bounds may vary for different loop classes.

- The bounds of the execution spaces, and consequently the concept of termination detection, are central aspects in this research area. The quality of the algorithm we utilize to find these bounds can only be determined with respect to a certain machine model, e.g., the signaling scheme [11] for distributed memory machines or the counter [1] and maximum scheme for shared memory machines. All methods compute a maximum — explicit or implicit — with different accuracy.

- Run time measurements are imperative to gather data on the differences between various shades of implementations of the (polyhedron) model.

  Maybe it is better to enumerate the target execution spaces less precisely if the (less exact) bounds can be computed faster.

- Virtually every decision that is made for implementing (certainly not only) the (polyhedron) model is accompanied by the trade-off between run time and memory consumption. One is always tempted to minimize *and* maximize, respectively, each aspect. However, this is not possible. We can only try to find a suited balance between run time and memory consumption.

  The main aim of parallelizing programs it to decrease run time. Thus, if we argue about the theoretical aspects of parallel execution, we should completely omit the aspect of memory, since nearly every attempt to save memory results in increased run time. Neglecting the question of memory consumption is also justified, because the availability (and affordability) of large amounts of fast memory is constantly increasing.

  If the parallelized program is executed on a real target machine, practical restrictions naturally force us to deviate from the theoretically possible performance. We have seen such a restriction earlier: there are not infinitely many processors. There is another one: there is not infinitely much memory. So it is certainly justified (and necessary) to think about the memory consumption of a certain implementation. However, this should not have the highest priority.

- We think that, in the end, every approach for exploiting parallelism has to face essentially the same problems. The polyhedron model offers only one way to face, learn about and solve these problems.

We hope that our research contributes some interesting and useful aspects to the examination of parallelism and to the development of automatic methods for exploiting existing parallelism in sequential programs.

Bibliography

[1] 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.

[2] A. Darte and F. Vivien. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. Research Report 96-06, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, April 1996.

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

[4] M. Dion and Y. Robert. Mapping Affine Loop Nests: New Results. *Lecture Notes in Computer Science 919*, pages 184–189. Springer Verlag, 1995.

[5] P. Faber. Target Output. Diploma thesis, Fakultät für Mathematik und Informatik, Universität Passau, 1997. In preparation.

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

[7] P. Feautrier. Dataflow Analysis of Array and Scalar References. *Int. J.Parallel Programming*, 20(1):23–53, February 1991

[8] U. Banerjee. Loop Transformations for Restructuring Compilers: The Foundations, pages 81–94. Kluwer, 1993.

[9] M. Griebl. The mechanical parallelization of loop nests containing while loops. Dissertation, Fakultät für Mathematik und Informatik, Universität Passau, 1996.

[10] M. Griebl and C. Lengauer. On the space-time mapping of WHILE-loops. *Parallel Processing Letters*, 4(3):221–232, September 1994.

[11] M. Griebl and C. Lengauer. A communication scheme for the distibuted execution of while loops. *Technical Report MIP-9406*, Fakultät für Mathematik und Informatik, Universität Passau, June 1994.

[12] M. Griebl and C. Lengauer. Classifying Loops for Space-Time Mapping. In L. Bougé, P. Fraigniaud, A. Mignotte and Y. Robert, editors, *EURO-PAR'96*, Lecture Notes in Computer Science 1123, volume I, pages 467–474, Springer-Verlag, 1996.

[13] M. Griebl and C. Lengauer. The Loop Parallelizer LooPo. In M. Gerndt, editor, *Proc. Sixth Workshop on Compilers for Parallel Computers*, Konferenzen des Forschungszentrums Jülich, 21:311–320. Forschungszentrum Jülich, 1996. URL: http://www.uni-passau.de/∼loopo/

[14] R. Guenz. The new LooPo scanner and parser. Internal Report, Fakultät für Mathematik und Informatik, Universität Passau, 1995.
URL: http://www.uni-passau.de/∼loopo/doc/guenz-p.ps.gz

[15] 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.

[16] H. Keimer. Datenabhängigkeitsanalyse zur Schleifenparallelisierung: Vergleich und Erweiterung zweier Ansätze. Diploma thesis, Fakultät für Mathematik und Informatik, Universität Passau, 1997.
URL: http://www.uni-passau.de/∼loopo/doc/keimer-d.ps.gz

[17] R. Kubias. Array Datenflußanalyse und Single Assignment Konversion. Diploma thesis, Fakultät für Mathematik und Informatik, Universität Passau, 1996.

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

[19] 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.

[20] T. G. Lewis and H. El-Rewini with In-Kyu Kim. Introduction to parallel computing. Prentice-Hall Inc., 1992

[21] W. Meisl. Practical Methods for Scheduling and Allocation in the Polytope Model. Diploma thesis, Fakultät für Mathematik und Informatik, Universität Passau, 1996.
URL: http://www.uni-passau.de/∼loopo/doc/meisl-d.ps.gz

[22] L. Rauchwerger and D. Padua. Parallelizing While Loops for Multiprocessor Systems. Proc. 9th Int. Parallel Programming Symposium (IPPS'95), pages 347–355, IEEE Computer Society Press, 1995.

[23] M. Schumergruber. Partitionierung von parallelen Schleifensätzen. Diploma thesis, Fakultät für Mathematik und Informatik, Universität Passau, 1997. In preparation.

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

[25] S. Wetzel. Automatic Code Generation in the Polyhedron Model. Diploma thesis, Fakultät für Mathematik und Informatik, Universität Passau, 1995.
URL: http://www.uni-passau.de/∼loopo/doc/wetzel-d.ps.gz

[26] C. Wieninger. Automatische Methoden zur Parallelisierung im Polyedermodell. Diploma thesis, Fakultät für Mathematik und Informatik, Universität Passau, 1995.
URL: http://www.uni-passau.de/∼loopo/doc/wieninger-d.ps.gz

[27] 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*.

[28] Y. Wu and T. G. Lewis. Parallelizing whlie loops. In H. D. Schwetman, editor, *Int. Conf. on Parallel Processing*, volume II, pages 1–8. CRC Press, 1990.