A Communication Scheme for the Distributed Execution of while Loops

Martin Griebl and Christian Lengauer

Fakultät für Mathematik und Informatik, Universität Passau,
D–94030 Passau, Germany, email: {griebl,lengauer}@fmi.uni-passau.de

Technical Report MIP-9406 / June 1994

Abstract. The mathematical model for the parallelization, or "space-time mapping", of loop nests is the polyhedron model. The presence of **while** loops in the nest complicates matters because the parallelized loop nest does not correspond to a polyhedron but instead to a subset that resembles a (multi-dimensional) comb. This comb can take on shapes that make the precise enumeration of its points by any parallel target loop nest impossible. We describe how to augment the target loop code to scan a finite superset of the comb and restrict execution of the loop body to the points of the comb.

Keywords: loop parallelization, parallelizing compilation, space-time mapping, **while** loop

1 Introduction

With the advent of increasingly large and affordable parallel computing systems, the topic of loop parallelization has recently received increased attention. Modeling loops with convex polyhedra has proved useful in characterizing and analyzing the space of possible parallelizations of sets of nested loops. This model has, so far, been applied only to loops all of whose bounds are known at compile time (**for** loops). Nests of **for** loops are modeled with polytopes, i.e., finite polyhedra. We present aspects of an extension of the model to loops whose upper bounds are not known before compile time (**while** loops).

After the indroduction of some notation (Sect. 2), we review briefly the polytope model and the parallelization of **for** loops in that model (Sect. 3). Polytopes are convex. The extension of the polytope model to **while** loops forces us to consider non-convex spaces (Sect. 4). We present a scheme for dealing with this lack of convexity (Sect. 5) and implement it as a communication skeleton for the parallel target loop nest (Sect. 6).

2 Notation

Our quantifier notation follows Dijkstra and Scholten [4]. Quantification over a dummy variable x is written $(Q\ x : R.x : P.x)$. Q is the quantifier, R is a predicate in x representing the range, and P is a term that depends on x. Formal logical deductions are given in the form:

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

where op is an operator from the set $\{\Leftarrow, \Leftrightarrow, \Rightarrow\}$.

A **while** loop is commonly written: **while** *condition* **do** *body* **end**. We view the **while** loop as a **for** loop with a dynamic upper bound, that is, we add an artificial **for** loop index that counts the number of executed iterations. Our **while** loop notation is: **for** *index* := 0 **while** *condition*(*index*) **do** *body* **end**, where *index* is a new variable that does not occur in the original loop. As in normal **for** loops, the increment of the loop index is implicit and follows the execution of the loop body. By supplying *index* as a parameter to *condition*, we make the dependence of *condition* on the **while** iteration explicit.

Note that every **while** loop can be represented this way.

3 for loops and the polytope model

Parallelization in the polytope model proceeds as follows [8].

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

Then, one applies an affine coordinate transformation $\mathcal{T}$, the "space-time mapping", to the polytope and obtains another polytope in which some dimensions enumerate space and the others enumerate time. (In the simplest case, there is only one time dimension.) We call the transformed polytope the *target space* and denote it with $\mathcal{TI}$. The standard techniques for space-time mapping in the polytope model require uniform dependences [7, 11, 12]. In this case, all points of $\mathcal{I}$ have identical dependence vectors that are independent of the problem size.

Finally, one translates this polytope back to a nest of target loops, where each space dimension becomes a parallel loop and each time dimension becomes a sequential loop. The target loop nest specifies asynchronous parallelism if the outer loops are the parallel ones, and synchronous parallelism if the outer loops are the sequential ones. When $\mathcal{T}$ is not unimodular, i.e., its inverse is not an integer matrix, $\mathcal{TI}$ contains "holes", i.e., it is not convex even though $\mathcal{I}$ is [1]. In this case, one has to take care that the target loops do not enumerate the holes. We call the process of enumerating the points of the target space *scanning*. Luckily, non-unimodular mappings distribute holes evenly throughout the target space. Therefore, there is always a target loop nest that scans $\mathcal{TI}$ precisely—whether $\mathcal{T}$ is unimodular [2] or not [15].

4 while loops and the polyhedron model

A loop nest with **while** loops is modeled by a polyhedron, not by a polytope. Since, at compile time, the upper bound of a **while** loop is unknown, the polyhedron is infinite in the corresponding dimension.

Also, a point of the polyhedron has a slightly different interpretation than in the **for** loop model: it represents not only one iteration step but also the boundary test preceding the step. One consequence is that an empty loop is modeled by one point, not by the empty set. Thus, any point of the index space that represents the first action of, say, loop w at level d stands for boundary test for all outer loops up to and including w (i.e., d tests) and for the first iteration of loop w, if any.

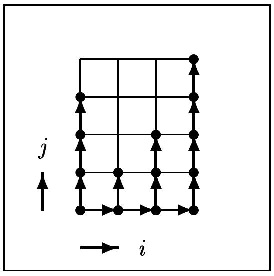

Fig. 1. A non-convex execution space with data dependences.

A nest with **while** loops executes only a subset of the infinite index space $\mathcal{I}$. We call this subset the *execution space* and name it $\mathcal{X}$. The dependences of points in $\mathcal{X}$ make it look like a (possibly, multi-dimensional) comb [6], as demonstrated in Fig. 1. Therefore, we call it also the *execution comb*. We give its transformation under the space-time mapping the name $\mathcal{TX}$.

In its pure form, the polyhedron model is restricted to uniform dependences [7, 11, 12]. The dependences introduced by a **while** index are not uniform—otherwise, the dependence graph would not be a comb but a grid. (For example, in Fig. 1, the i dependences are only present for one value of j.) One can view any affine space-time mapping of the comb as a restriction of the same mapping on the comb's "uniform closure". In a uniform closure, one adds arrows to make any non-uniform dependence uniform. The affinity of the mapping ensures that the added arrows do not violate the mapping's consistency—that is, the restriction of the mapping derived for the uniform closure to the comb is exactly the mapping we want. This gives us the right to make use of the theory for **for** loop parallelization.

If only the outermost loop of the nest is a **while** loop, the execution space of the nest is still convex and, thus, can be scanned precisely. But if we allow at least one **while** loop properly inside the nest, only space-time mappings that fulfill certain conditions yield a precisely scannable execution space [6]. For example, consider Fig. 2: there is no nest of target loops that scans the points without encountering the hole at coordinates $(3, 2)$.

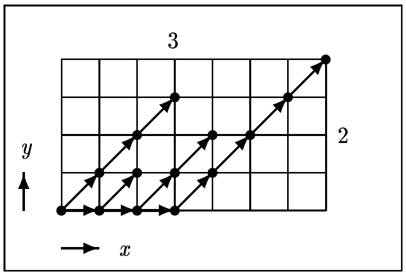

Fig. 2. Unscannable transformed execution space with data dependences.

One more remark is in order. In the source loop nest, we set the lower bound of a **while** loop to zero, for any level r. In the transformed execution space, this bound may be mapped to an arbitrary integer lb_r.

Points in the execution space are d-dimensional but, to save space, we abbreviate the representation of a point at the base of the multi-dimensional comb in dimension r from $(p_1, \cdots, p_r, lb_{r+1}, \cdots, lb_d)$ to $(p_1, \cdots, p_r)$.

5 Controlling the scan of an unscannable execution space

We call an execution space that cannot be scanned by any target loop nest without encountering holes *unscannable*. Scanning an unscannable execution space $\mathcal{TX}$ with some fixed target loop nest yields a proper superset of $\mathcal{TX}$; we name this set $\mathcal{TS}$ and its inverse image under the space-time mapping $\mathcal{S}$. In this section, we define a logical predicate that is an accurate recognizer of the points in $\mathcal{TX}$, i.e., that distinguishes the points in $\mathcal{TX}$ from those outside. In the following section, we use this predicate to prevent the execution of holes in the target polyhedron at run time.

Definition 1 (Activity recognizer *active$_r$*). Let r by some level of the source loop nest and w the **while** loop at that level. $active_r$ holds for any point p in $\mathcal{I}$ iff the source program enumerates p, that is, iff at least the **while** condition $cond_r$ of loop w is evaluated at point p. Formally:

$$
\begin{array}{ll}
(\forall\ (p_1,\cdots,p_d)\ :\ (p_1,\cdots,p_d)\in\mathcal{I}\ :\ (\forall\ r: 1\leq r\leq d\ :\ active_r(p_1,\cdots,p_r) = & \\
\quad \textbf{if}\ p_r > lb_r \quad\quad\quad \rightarrow active_r(p_1,\cdots,p_r-1) \wedge cond_r(p_1,\cdots,p_r-1) & (1)\\
\quad [\!]\ \ p_r = lb_r \wedge r > 1 \rightarrow active_{r-1}(p_1,\cdots,p_{r-1}) \wedge cond_{r-1}(p_1,\cdots,p_{r-1}) & (2)\\
\quad [\!]\ \ p_r = lb_r \wedge r = 1 \rightarrow \text{true} & (3)\\
\quad [\!]\ \ p_r < lb_r \quad\quad\quad \rightarrow \text{false} & (4)\\
\quad \textbf{fi}\ )) &
\end{array}
$$

The alternatives of the defining equation can be explained as follows. Point $(p_1, \cdots, p_r)$ is active with respect to level r iff

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

(2) the point represents the first step of an inner loop, the **while** condition holds for the immediately enclosing loop and the point is active with respect to the level of that loop, or
(3) the point represents the first step of the entire loop nest.

In all other cases, $(p_1, \cdots, p_r)$ is inactive with respect to level r. These include the case, where the **while** condition of w is violated (covered by alternatives (1) and (2)), and the case that the point is not even in the index space (alternative (4)). Note, that points for which the **while** condition holds at level r but not at level $r+1$ are active with respect to level r but not with respect to levels $r+1$ and deeper.

The recursive definition of predicate $active_r$ follows the dependences that are introduced by the **while** indices. Since our space-time mapping must respect these dependences, we can be sure that, during scanning, the activity of any point p in $\mathcal{TI}$ need not be checked before the activity of its predecessor has been checked. Therefore, we can compute predicate $active_r$ for every point on every tooth of the execution comb, in sequence from the root to the tip, and store the result until it is needed.

Note that $active_r$ at point $(p_1, \cdots, p_r)$ depends on $active_r$ at exactly one other point (see the definition); $active_r$ at point $(p_1, \cdots, p_r)$ is used for the computation of $active_r$ at point $(p_1, \cdots, p_r+1)$ and, if r is not the innermost loop level, also for the computation of $active_{r+1}$ at itself.

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

Definition 2 (Recognizer *executed$_r$* and *executed*).

$$(\forall\, r \,:\, 1 \leq r \leq d \,:\, (\forall\, p \,:\, p \in \mathcal{I} \,:\, executed_r(p_1, \cdots, p_r) \Leftrightarrow \\ (active_r(p_1, \cdots, p_r) \wedge condition_r(p_1, \cdots, p_r))))$$

$$(\forall\, p \,:\, p \in \mathcal{I} \,:\, executed(p) \Leftrightarrow executed_d(p_1, \cdots, p_d))$$

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

Definition 3 (Execution space). $\mathcal{X} = \{p \in \mathcal{I} \,:\, executed(p)\}$

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

Definition 4 (Activity space). $\widehat{\mathcal{X}} = \{p \in \mathcal{I} \,:\, (\exists\, r \,:\, 1 \leq r \leq d \,:\, active_r(p))\}$

6 Termination Detection

The most difficult problem in the derivation of target loop code is the determination of the upper bound for the outermost loop (if the source contains at least one inner **while** loop). The following example demonstrates that even the simplest transformation may cause trouble.

Example 1. Take the following loop nest:

for $i := 0$ **to** 3 **do**
  for $j := 0$ **while** *condition*(i, j) **do**
    body
  end
end

and as space-time mapping the identity $(i, j) \mapsto (p, t)$, where p is space and t is time. (We assume that this transformation respects the dependences of the body. The **while** dependence is satisfied.) With this mapping, every processor executes the **while** loop for a fixed value of i. Clearly, each processor can determine the condition for its termination locally, but no processor can detect the termination of any other processor, unless additional measures are taken. Say some statement follows the parallel loop nest:

parfor $p := 0$ **to** 3 **do**
  seqfor $t := 0$ **while** *condition*(p, t) **do**
    body
  end
end
statement

How do we know when to start the execution of *statement*?

Here, we choose the asynchronous case (where the time loop is innermost), but the same problem occurs in the synchronous case (where the time loop is outermost): there is no elegant way of expressing the termination condition of the outermost loop. In this example, it is given by

$$(\forall\, p \,:\, 0 \le p \le 3 \,:\, (\exists\, t' \,:\, 0 \le t' \le t \,:\, \neg\mathit{condition}(p, t'))).$$

These quantifications are potentially costly because, in general, there is no bound on their ranges.

We present a solution of the termination problem that requires only communication between neighboring processors.

6.1 Idea

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

If we ensure that no tooth terminates before it has been informed of the termination of its neighbors, points that are involved in these communications are partially maximal until the point $(p_1, \cdots, p_d)$ in $\mathcal{S}$ is reached whose coordinates are all maximal, i.e., have the property $(\forall\, p' \,:\, p' \in \mathcal{X} \,:\, (\forall\, r \,:\, 1 \le r \le d \,:\, p'_r \le p_r))$. When scanning this point, we can terminate all target loops.

The propagation of the maxima, up to depth r, proceeds by valueless signals. Signal $sig_k^r(p_1, \cdots, p_r)$ starts at point $(p_1, \cdots, p_r)$ at depth r and is sent to the neighboring tooth in direction k, where k is some outer level with respect to r, i.e., $k < r$.

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

Example 2. Consider the three-dimensional comb of Fig. 3.

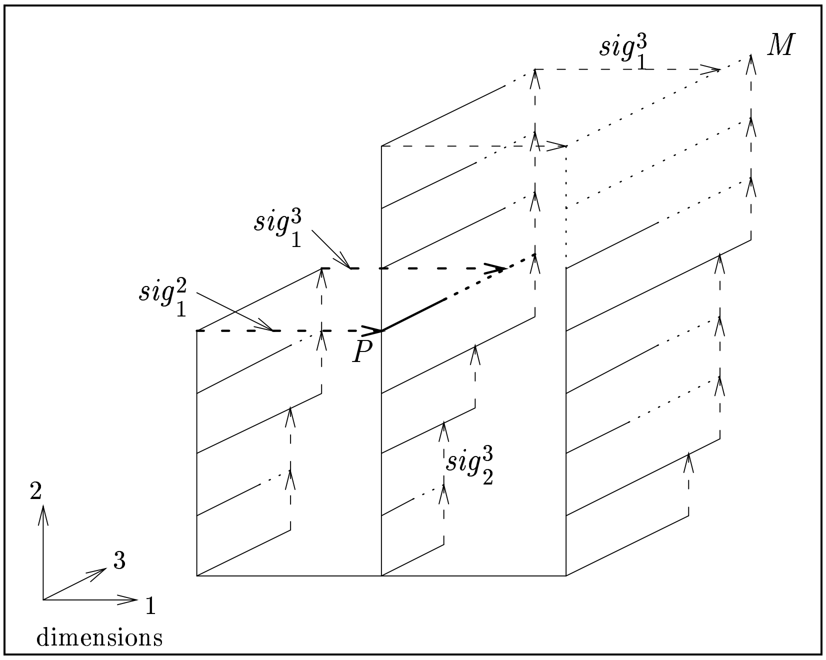

Fig. 3. A three-dimensional comb.

Our scheme is more easily understood in the synchronous model.

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

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

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

$$sig_1^2(p_1,p_2) \Rightarrow (\forall\, p_1',p_2' \,:\, (p_1',p_2')\in\mathcal{X} \wedge p_1'\leq p_1 \,:\, p_2'\leq p_2)$$

The determination of the maximal depth of teeth in each vertical plane (p_1 constant) proceeds analogously. Signals sig_2^3 are sent from every (perhaps extended, since waiting) tooth τ along dimension 3 to its upper neighbor of that plane, indicating that the current depth (the length of τ) is maximal for all teeth in dimension 3 to the left and including τ, for fixed p_1. Formally:

$$sig_2^3(p_1,p_2,p_3) \Rightarrow (\forall\, p_2',p_3' \,:\, (p_1,p_2',p_3')\in\mathcal{X} \wedge p_2'\leq p_2 \,:\, p_3'\leq p_3)$$

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

$$sig_1^3(p_1,p_2,p_3) \Rightarrow (\forall\, p_1',p_2',p_3' \,:\, (p_1',p_2',p_3')\in\mathcal{X} \wedge p_1'\leq p_1 \,:\, p_2'\leq p_2 \wedge p_3'\leq p_3)$$

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

6.2 Formalization

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

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

Definition 5 (Partial maximality m_k^r). $m_k^r(p_1,\cdots,p_r)$ iff for fixed indices at levels 1 to $k-1$ and for points $(p_1',\cdots,p_r')$ below $(p_1,\cdots,p_r)$ at level k ($p_k'\leq p_k$), point $(p_1,\cdots,p_r)$ is maximal in all dimensions $k+1,\cdots,r$. Formally:

$$
\begin{aligned}
&(\forall\, r\ : 0 < r \le d\ :\ (\forall\ (p_1,\cdots,p_r) : (p_1,\cdots,p_r) \in \widehat{\mathcal{X}}\ :\\
&\qquad (\forall\ k\ :\ 0 \le k < r\ :\ m_k^r(p_1,\cdots,p_r)\ =\\
&\qquad\qquad (\forall\ p'_k,\cdots,p'_r\ :\ (p_1,\cdots,p_{k-1},p'_k,\cdots,p'_r) \in \widehat{\mathcal{X}} \wedge p'_k \le p_k\ :\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad p'_{k+1} \le p_{k+1},\cdots,p'_r \le p_r))))
\end{aligned}
$$

If $m_k^r(p_1,\cdots,p_r)$, we call point $(p_1,\cdots,p_r)$ *partially maximal* with respect to dimensions $k+1$ to r.

Note that, for $k=0$, the right-hand side of Def. 5 simplifies to:

$$
(\forall\ p'_1,\cdots,p'_r\ :\ (p'_1,\cdots,p'_r) \in \widehat{\mathcal{X}}\ :\ p'_1 \le p_1 \wedge \cdots \wedge p'_r \le p_r)
$$

For our communication scheme of m by signals, we need an additional predicate, $w_k^r(p_1,\cdots,p_r)$, which indicates that the tooth that is rooted at $(p_1,\cdots,p_{r-1})$ and extends along dimension r must wait until signal sig_k^r arrives at some point $(p_1,\cdots,p_{r-1},p'_r)$ with $p'_r \ge p_r$. These additional points at which a tooth waits but executes nothing make the difference between $\mathcal{S}$ and $\mathcal{X}$. The lemmata that follow are valid for all points in $\mathcal{S}$.

Definition 6 (*sig* and *w*).

$$
\begin{aligned}
&(\forall\ (p_1,\cdots,p_d)\ :\ (p_1,\cdots,p_d) \in \mathcal{I}\ :\ (\forall\ k,r\ :\ 0 \le k < r \le d\ :\ sig_k^r(p_1,\cdots,p_r)\ =\\
&\neg executed_r(p_1,\cdots,p_r) \wedge\\
&\quad (\forall\ s\ :\ 1 \le s < r\ :\ \neg w_s^r(p_1,\cdots,p_r) \vee sig_s^r(p_1,\cdots,p_s-1,\cdots,p_r)) \wedge\\
&\quad (r > k+1 \Rightarrow sig_k^{r-1}(p_1,\cdots,p_{r-1}))\ ))
\end{aligned}
$$

For all other points *sig* is initialized with false:

$$
\begin{aligned}
&(\forall\ (p_1,\cdots,p_d)\ :\ (p_1,\cdots,p_d) \in \mathbb{Z}^d - \mathcal{I}\ :\ (\forall\ k,r\ :\ 0 \le k < r \le d\ :\\
&\quad sig_k^r(p_1,\cdots,p_r)\ = \text{false}))
\end{aligned}
$$

$$
\begin{aligned}
&(\forall\ (p_1,\cdots,p_d)\ :\ (p_1,\cdots,p_d) \in \mathcal{I}\ :\ \ (\forall\ k,r\ :\ 0 \le k < r \le d\ :\ w_k^r(p_1,\cdots,p_r)\ =\\
&\quad \textbf{if}\ k=0 \qquad\qquad\qquad\qquad\qquad \to \text{false} && (1)\\
&\quad []\ r=k+1 \wedge k>0 \wedge p_{k+1}=lb_{k+1} \to p_k \ne lb_k && (2)\\
&\quad []\ r>k+1 \wedge k>0 \wedge p_{k+1}=lb_{k+1} \to sig_k^{r-1}(p_1,\cdots,p_k-1,\cdots,p_{r-1}) && (3)\\
&\quad []\ r>k+1 \wedge k>0 \wedge p_{k+1}>lb_{k+1} \to && (4)\\
&\qquad\quad w_k^r(p_1,\cdots,p_{r-1},p_r-1)\ \wedge\ \neg sig_k^r(p_1,\cdots,p_k-1,\cdots,p_{r-1},p_r-1)\\
&\quad \textbf{fi}\ ))
\end{aligned}
$$

These equations can be explained as follows.

sig states that any point of a tooth that need not be executed and that does not have to wait for any signal sends signal sig_k^r if either the tooth and the signal lie in a two-dimensional plane (recursion base) or the root $(p_1,\cdots,p_{r-1})$ of the tooth has already sent the signal into the same direction (recursion).

$w_k^r(p_1,\cdots,p_r)$ states whether the point $(p_1,\cdots,p_r)$ at level r has to wait for some signal from direction k:

(1) No point has to wait for signals from the host.
(2) In every two-dimensional subspace (dimensions k and $r = k{+}1$), every tooth, i.e., at least the first point of it (with $p_{k+1} = lb_{k+1}$), has to wait for a signal from the tooth immediately preceding it—if any, i.e., if $p_k \neq lb_k$.
(3) In every at least three-dimensional subspace (dimensions k to $r > k+1$), every tooth parallel to dimension r, i.e., at least the first point of it, has to wait for some signal from direction k iff its root at level $r-1$ has received a signal from the same direction k.
(4) Any point of a tooth that is not the first point has to wait for a signal iff its predecessor on the tooth had to wait and did not receive the signal it was waiting for.

6.3 Signals and their significance for local maximality

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

Lemma 7 (Local maximum). *A point of some tooth along dimension r that need not be executed with respect to r and need not wait for a signal is maximal with respect to dimension r. Formally:*

$$\begin{array}{l}(\forall\ (p_1,\cdots,p_r)\ :\ (p_1,\cdots,p_r)\in\mathcal{S}\ :\ \neg executed_r(p_1,\cdots,p_r)\ \wedge\\ \quad (\forall\ s\ :\ 1\leq s<r\ :\ (\neg w^r_s(p_1,\cdots,p_r)\vee sig^r_s(p_1,\cdots,p_s-1,\cdots,p_r)))\Rightarrow\\ \quad (\forall\ p'_r\ :\ (p_1,\cdots,p_{r-1},p'_r)\in\mathcal{S}\ :\ p'_r\leq p_r))\end{array}$$

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

$$\begin{array}{ll} & \neg(\forall\ p'_r\ :\ (p_1,\cdots,p_{r-1},p'_r)\in\mathcal{S}\ :\ p'_r\leq p_r)\\ \Leftrightarrow & \{\ \text{predicate calculus}\ \}\\ & (\exists\ p'_r\ :\ (p_1,\cdots,p_{r-1},p'_r)\in\mathcal{S}\ :\ p'_r>p_r)\\ \Rightarrow & \{\ \text{if point } (p_1,\cdots,p_{r-1},p'_r) \text{ is scanned, it must be active or waiting}\ \}\\ & (\exists\ p'_r\ :\ (p_1,\cdots,p_{r-1},p'_r)\in\mathcal{S}\ :\ active_r(p_1,\cdots,p_{r-1},p'_r)\ \vee\\ & \quad (\exists\ s\ :\ 1\leq s<r\ :\ w^r_s(p_1,\cdots,p_{r-1},p'_r)\wedge\neg sig^r_s(p_1,\cdots,p_s-1,\cdots,p'_r)))\\ \Rightarrow & \{\ \text{predicate calculus}\ \}\\ & (\exists\ p'_r\ :\ (p_1,\cdots,p_{r-1},p'_r)\in\mathcal{S}\ :\ active_r(p_1,\cdots,p_{r-1},p'_r)\ \vee\\ & \quad condition_r(p_1,\cdots,p_{r-1},p'_r)\ \vee\\ & \quad (\exists\ s\ :\ 1\leq s<r\ :\ w^r_s(p_1,\cdots,p_{r-1},p'_r)\wedge\neg sig^r_s(p_1,\cdots,p_s-1,\cdots,p'_r)))\\ \Leftrightarrow & \{\ \text{predicate calculus}\ \}\\ & \neg(\forall\ \ (p_1,\cdots,p_r)\ :\ (p_1,\cdots,p_r)\in\mathcal{S}\ :\\ & \quad \neg(active_r(p_1,\cdots,p_r)\ \wedge\ condition_r(p_1,\cdots,p_r))\ \wedge\\ & \quad (\forall\ s\ :\ 1\leq s<r\ :\ \neg w^r_s(p_1,\cdots,p_r)\vee sig^r_s(p_1,\cdots,p_s-1,\cdots,p_r)))\\ \Leftrightarrow & \{\ \text{Def. 2}\ \}\\ & \neg(\forall\ \ (p_1,\cdots,p_r)\ :\ (p_1,\cdots,p_r)\in\mathcal{S}\ :\ \neg executed_r(p_1,\cdots,p_r)\ \wedge\\ & \quad (\forall\ s\ :\ 1\leq s<r\ :\ \neg w^r_s(p_1,\cdots,p_r)\vee sig^r_s(p_1,\cdots,p_s-1,\cdots,p_r)))\end{array}$$

In the following lemma, $\mathcal{SUB}^r_k(p_1,\cdots,p_{k-1})$ is the subspace of $\mathcal{S}$ in dimensions k to r $(1\leq k<r\leq d)$ and at fixed coordinates $(p_1,\cdots,p_{k-1})$.

Lemma 8 (sig implements m).

$$(\forall\, k,r \,:\, 0 \le k < r \le d \,:\, (\forall\, (p_1,\cdots,p_r) \,:\, (p_1,\cdots,p_r) \in \mathcal{S} \,:\, \\ sig_k^r(p_1,\cdots,p_r) \Rightarrow m_k^r(p_1,\cdots,p_r)))$$

Proof. By induction on the dimension of $\mathcal{SUB}_k^r(p_1,\cdots,p_{k-1})$.

- Induction base ($r = k+1$):
  For $sig_k^{k+1}(p_1,\cdots,p_{k+1}) \Rightarrow m_k^{k+1}(p_1,\cdots,p_{k+1})$ we must distinguish two cases, since p_0 does not exist.
  - First case: $k = 0$

$$\begin{array}{ll} & sig_0^1(p_1) \\ \Leftrightarrow & \{\text{ definition of } sig \text{ with } k=0,\ r=1 \ \} \\ & \neg executed_1(p_1) \\ \Rightarrow & \{\text{ Lemma 7 with } k=0,\ r=1 \ \} \\ & (\forall\, p_1' \,:\, p_1' \in \mathcal{S} \,:\, p_1' \le p_1) \\ \Leftrightarrow & \{\text{ definition of } m_0^1 \ \} \\ & m_0^1(p_1) \end{array}$$

  - Second case: $k > 0$
    We prove $sig_k^{k+1}(p_1,\cdots,p_{k+1}) \Rightarrow m_k^{k+1}(p_1,\cdots,p_{k+1})$ by induction on p_k.
    - Induction base ($p_k = lb_k$):

$$\begin{array}{ll} & sig_k^{k+1}(p_1,\cdots,p_{k+1}) \\ \Leftrightarrow & \{\text{ definition of } sig \text{ with } p_k = lb_k \\ & \quad \text{and simplification for } r = k+1 \ \} \\ & \neg executed_{k+1}(p_1,\cdots,p_{k+1}) \ \wedge (\forall\, s \,:\, 1 \le s < k+1 \,: \\ & \qquad \neg w_s^{k+1}(p_1,\cdots,p_{k+1}) \vee sig_s^{k+1}(p_1,\cdots,p_s-1,\cdots,p_{k+1})) \\ \Rightarrow & \{\text{ Lemma 7 } \} \\ & (\forall\, p_{k+1}' \,:\, (p_1,\cdots,p_k,p_{k+1}') \in \mathcal{S} \,:\, p_{k+1}' \le p_{k+1}) \\ \Leftrightarrow & \{\ lb_k \text{ is the smallest } p_k; \\ & \quad \text{introduction of a new dummy with empty range } \} \\ & (\forall\, p_k', p_{k+1}' \,:\, (p_1,\cdots,p_{k-1},p_k',p_{k+1}') \in \mathcal{S} \ \wedge\ p_k' \le p_k \,: \\ & \qquad p_{k+1}' \le p_{k+1}) \\ \Leftrightarrow & \{\text{ definition of } m \ \} \\ & m_k^{k+1}(p_1,\cdots,p_{k+1}) \end{array}$$

    - Induction step ($p_k - 1 \to p_k$, where $p_k > lb_k$):

$$\begin{array}{ll} & sig_k^{k+1}(p_1,\cdots,p_{k+1}) \\ \Leftrightarrow & \{\text{ definition of } sig \text{ with } p_k > lb_k \\ & \quad \text{and simplification for } r = k+1 \ \} \\ & \neg executed_{k+1}(p_1,\cdots,p_{k+1}) \ \wedge (\forall\, s \,:\, 1 \le s < k+1 \,: \\ & \qquad (\neg w_s^{k+1}(p_1,\cdots,p_{k+1}) \vee sig_s^{k+1}(p_1,\cdots,p_s-1,\cdots,p_{k+1}))) \\ \Leftrightarrow & \{\text{ duplication of } \neg w_k^{k+1} \vee sig_k^{k+1}(p_1,\cdots,p_{k-1},p_k-1,p_{k+1}) \ \} \\ & \neg executed_{k+1}(p_1,\cdots,p_{k+1}) \ \wedge (\forall\, s \,:\, 1 \le s < k+1 \,: \\ & \qquad (\neg w_s^{k+1}(p_1,\cdots,p_{k+1}) \vee sig_s^{k+1}(p_1,\cdots,p_s-1,\cdots,p_{k+1}))) \ \wedge \\ & \qquad \neg w_k^{k+1}(p_1,\cdots,p_{k+1}) \vee sig_k^{k+1}(p_1,\cdots,p_{k-1},p_k-1,p_{k+1}) \\ \Rightarrow & \{\text{ Lemma 7 } \} \end{array}$$

$$
\begin{array}{ll}
 & (\forall\, p'_{k+1} \,:\, (p_1,\cdots,p_k,p'_{k+1}) \in \mathcal{S} \,:\, p'_{k+1} \leq p_{k+1}) \;\wedge \\
 & \qquad \neg w_k^{k+1}(p_1,\cdots,p_{k+1}) \vee sig_k^{k+1}(p_1,\cdots,p_{k-1},p_k-1,p_{k+1}) \\
\Rightarrow & \{ \text{ by structural induction on the definition of } w \ \} \\
 & (\forall\, p'_{k+1} \,:\, (p_1,\cdots,p_k,p'_{k+1}) \in \mathcal{S} \,:\, p'_{k+1} \leq p_{k+1}) \;\wedge\; (p_k = lb_k \;\vee \\
 & \qquad (\exists\, \widehat{p_{k+1}} \,:\, \widehat{p_{k+1}} \leq p_{k+1} \,:\, sig_k^{k+1}(p_1,\cdots,p_k-1,\widehat{p_{k+1}}))) \\
\Leftrightarrow & \{ \ p_k \neq lb_k \text{ (induction step) } \} \\
 & (\forall\, p'_{k+1} \,:\, (p_1,\cdots,p_k,p'_{k+1}) \in \mathcal{S} \,:\, p'_{k+1} \leq p_{k+1}) \;\wedge \\
 & \qquad (\exists\, \widehat{p_{k+1}} \,:\, \widehat{p_{k+1}} \leq p_{k+1} \,:\, sig_k^{k+1}(p_1,\cdots,p_k-1,\widehat{p_{k+1}})) \\
\Rightarrow & \{ \text{ induction hypothesis for } sig_k^{k+1}(p_1,\cdots,p_k-1,\widehat{p_{k+1}}) \ \} \\
 & (\forall\, p'_{k+1} \,:\, (p_1,\cdots,p_k,p'_{k+1}) \in \mathcal{S} \,:\, p'_{k+1} \leq p_{k+1}) \;\wedge \\
 & \qquad (\exists\, \widehat{p_{k+1}} \,:\, \widehat{p_{k+1}} \leq p_{k+1} \,:\, m_k^{k+1}(p_1,\cdots,p_k-1,\widehat{p_{k+1}})) \\
\Leftrightarrow & \{ \ \widehat{p_{k+1}} \leq p_{k+1}; \text{ definition of } m) \ \} \\
 & (\forall\, p'_{k+1} \,:\, (p_1,\cdots,p_k,p'_{k+1}) \in \mathcal{S} \,:\, p'_{k+1} \leq p_{k+1}) \;\wedge \\
 & \qquad m_k^{k+1}(p_1,\cdots,p_k-1,p_{k+1}) \\
\Leftrightarrow & \{ \text{ definition of } m \ \} \\
 & (\forall\, p'_{k+1} \,:\, (p_1,\cdots,p_k,p'_{k+1}) \in \mathcal{S} \,:\, p'_{k+1} \leq p_{k+1}) \;\wedge (\forall\, p'_k,p'_{k+1} \,: \\
 & \qquad (p_1,\cdots,p_{k-1},p'_k,p'_{k+1}) \in \mathcal{S} \;\wedge\; p'_k \leq p_k - 1 \,:\, p'_{k+1} \leq p_{k+1}) \\
\Leftrightarrow & \{ \text{ combination of two quantification ranges } \} \\
 & (\forall\, p'_k,p'_{k+1} \,:\, (p_1,\cdots,p_{k-1},p'_k,p'_{k+1}) \in \mathcal{S} \;\wedge\; p'_k \leq p_k \,:\, p'_{k+1} \leq p_{k+1}) \\
\Leftrightarrow & \{ \text{ definition of } m \ \} \\
 & m_k^{k+1}(p_1,\cdots,p_{k+1})
\end{array}
$$

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

$$
\begin{array}{ll}
 & sig_{k-1}^r(p_1,\cdots,p_r) \\
\Leftrightarrow & \{ \text{ definition of } sig \text{ with } r > k+1 \ \} \\
 & \neg executed_r(p_1,\cdots,p_r) \;\wedge\; (\forall\, s \,:\, 1 \leq s < r \,: \\
 & \qquad \neg w_s^r(p_1,\cdots,p_r) \vee sig_s^r(p_1,\cdots,p_s-1,\cdots,p_r)) \;\wedge \\
 & \qquad sig_{k-1}^{r-1}(p_1,\cdots,p_{r-1}) \\
\Rightarrow & \{ \text{ Lemma 7 } \} \\
 & (\forall\, p'_r \,:\, (p_1,\cdots,p_{r-1},p'_r) \in \mathcal{S} \,:\, p'_r \leq p_r) \;\wedge\; sig_{k-1}^{r-1}(p_1,\cdots,p_{r-1}) \\
\Rightarrow & \{ \text{ induction hypothesis for } sig_{k-1}^{r-1} \ \} \\
 & (\forall\, p'_r \,:\, (p_1,\cdots,p_{r-1},p'_r) \in \mathcal{S} \,:\, p'_r \leq p_r) \;\wedge\; m_{k-1}^{r-1}(p_1,\cdots,p_{r-1}) \\
\Leftrightarrow & \{ \text{ definition of } m \ \} \\
 & (\forall\, p'_r \,:\, (p_1,\cdots,p_{r-1},p'_r) \in \mathcal{S} \,:\, p'_r \leq p_r) \;\wedge \\
 & \qquad (\forall\, p'_{k-1},\cdots,p'_{r-1} \,:\, (p_1,\cdots,p_{k-2},p'_{k-1},\cdots,p'_{r-1}) \in \mathcal{S} \;\wedge\; p'_{k-1} \leq p_{k-1} \,: \\
 & \qquad p'_k \leq p_k \wedge \cdots \wedge p'_{r-1} \leq p_{r-1}) \\
\Rightarrow & \{ \text{ combination of two quantification ranges } \} \\
 & (\forall\, p'_k,\cdots,p'_r \,:\, (p_1,\cdots,p_{k-2},p'_{k-1},\cdots,p'_r) \in \mathcal{S} \;\wedge\; p'_{k-1} \leq p_{k-1} \,: \\
 & \qquad p'_k \leq p_k \wedge \cdots \wedge p'_r \leq p_r) \\
\Leftrightarrow & \{ \text{ definition of } m \ \} \\
 & m_{k-1}^r(p_1,\cdots,p_r)
\end{array}
$$

Our aim has been to identify the point of $\mathcal{S}$ with maximal coordinates in all dimensions. The scanning of this point indicates the termination of the entire target loop nest. We have constructed a signaling scheme in which this point sends signal sig_0^d to the host.

Remark 9.

- A tooth in direction r must send at least one signal, into direction $r-1$.
- One simple optimization can be made immediately: signals need not be sent to points p along a tooth that has terminated, i.e., points that neither are active nor wait for a signal. Thus, $sig_k^r(p_1,\cdots,p_r)$ implies that, for any k' with $k'<k$, $sig_{k'}^r(p_1,\cdots,p_r)$ need not be sent. In Example 2, we have already omitted these signals, but they are, of course, part of Def. 6. Thus, every terminating tooth has to send at most one signal.
- In summary, every tooth must send exactly one signal when it terminates.

Future work will reduce the size of $\mathcal{S}$ by extracting tighter individual termination conditions for each target loop instead of one common termination condition for all loops.

7 The target code

There are standard techniques with which a target polyhedron can be transformed into a parallel loop nest [8]. In this section, we present an augmentation of the target loop body such that the irregular shape of the transformed execution space is dealt with properly. Our augmentation implements the signaling scheme presented in the previous section. We prove that, while the target loop bounds coincide with the boundaries of $\mathcal{TS}$, the augmented body reduces the execution of the loop nest to precisely the points of $\mathcal{TX}$.

We present the target code in several stages. We start with a target program that is still abstract in the following sense: (1) it is single-assignment, and (2) it does not specify whether the parallelism is synchronous or asynchronous. Next, we explain how to introduce re-assignments into the target program by a simple transformation. Finally, we discuss the differences between synchrony and asynchrony in the parallel execution.

7.1 The single-assignment code

The target code must state communications (send and receive primitives) explicitly. We use three primitives that are executable on the PARIX operating system [10], one for transmitting and two for receiving. The reception mode depends on the type of message transmitted.

Our signaling scheme contains two types of message.

- One type is responsible for the propagation of information along one tooth. An example is the value of the predicate *active*. The receipt of a message of this type is necessary for the execution of the loop body at the respective point.

– The other type of messages is for signals *sig*. These signals must be "probed" [9], since execution of a loop iteration may proceed without their receipt.

Let us briefly describe the three primitives:

– **asend**(*dir*, *list_of_vals*) is a PARIX command that transmits a list of values into a specified direction in $(d-1)$-dimensional space. Communication is asynchronous; this makes our code generation scheme independent of the space-time mapping, as discussed further in the conclusions.
– **receive**(*dir*, *list_of_vals*) is a PARIX command that performs a blocking receive of a list of values from a specified direction.
– **creceive**(*dir*, *list_of_vals*) is a command defined by us that performs a non-blocking receive of a list of values from a specified direction. Martin [9] defines the value of a probe $\overline{Y}$ on some communication action Y as a boolean value that indicates whether the corresponding communication action is pending. In our context, Y is a **receive** command from some direction and, thus, $\overline{Y}$ holds iff the corresponding **asend** command did already take place. With this construct, we define **creceive**(*dir*, *list_of_vals*) as follows:

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

Note: $\overline{\textbf{receive}(dir, list\_of\_vals)}$ corresponds roughly to the PARIX command **Select**(**ReceiveOption**(*dir*)) [10].

We are now able to present the single assignment target program. For simplicity, we implement termination by the PARIX command **AbortServer** that is executed only by an iteration that is maximal for all dimensions (i.e., partially maximal with respect to dimensions 1 to d, as defined in Def. 5). Thus, the target **while** loops need no upper bound.

The target loop nest is displayed in Fig. 4 and refined in the subsequent Figs. 5–6. This code is for asynchronous parallel execution on distributed-memory machines. The necessary changes for synchronous execution and/or shared-memory are discussed in Sect. 7.3.

The indicated augmentation of the loop body for every level r, i.e., $b_r(p_1, \cdots, p_r)$, consists of two parts:

– The prefix $b_r^{\rm pre}(p_1, \cdots, p_r)$, executed before the transformed source loop body, is displayed in Fig. 5. It receives all necessary data and calculates the output values of all variables that are introduced by our signaling scheme.
– The postfix $b_r^{\rm post}(p_1, \cdots, p_r)$, executed after the source loop (at least for asynchronous execution, see Remark 10), is displayed in Fig. 6. It is responsible for sending all necessary information.

Let us now discuss the code in detail:

Variables:

– $chan_{\rm S}^k$ represents the channel for the signals that travel along dimension k. $chan_{\rm D}^r$ represents the channel for the data, i.e. $targ\_active_r$, k_r, and w_k^r, $1 \leq k < r$, that are propagated along every tooth.

```
k_1(lb_1) := 0
targ_active_1(lb_1) := true
for q_1 := lb_1 while true do
⋮
    for q_d := lb_d while true do
        b_1^pre(p_1)
        ⋮
        b_d^pre(p_1, ⋯ , p_d)
        if targ_active_d(p_1, ⋯ , p_d) and condition_d(p_1, ⋯ , p_d) then
            b(p_1, ⋯ , p_d)
        fi
        b_1^post(p_1)
        ⋮
        b_d^post(p_1, ⋯ , p_d)
    end
⋮
end
```

Fig. 4. The target loop nest

- k_r is the direction in which the tooth must send a signal before it terminates. It corresponds to the lower index k of $sig_k^r(p_1, \cdots, p_r)$. The optimization outlined in Remark 9 ensures that there is a unique k for each tooth.
- $sigval_r$ is equal to sig_k^r of Def. 6.
- w_k^r is as in Def. 6.
- $targ\_active_r(\mathcal{T}p)$ is the counterpart of the predicate *active* in the source loop nest, i.e., for every point p of $\mathcal{I}$, the value of $targ\_active_r(\mathcal{T}p)$ at the end of the program is equal to the value of $active_r(p)$. We say *targ_active implements* predicate *active* and prove this fact in Lemma 14. The value of $targ\_active(\mathcal{T}p)$ is undefined if $\mathcal{T}p$ is not scanned by the target loop nest.

Execution:

- The outermost **if** clause prevents the body from receiving, calculating and sending signals and messages that are not specified by our signaling scheme of Def. 6, i.e., signals at depth $r' > r$.
- Part A probes signals that are expected and receives those that are actually being sent. Thus, $rcved_k^r(p_1, \cdots, p_r)$ is equal to $sig_k^r(p_1, \cdots, p_k - 1, \cdots, p_r)$.
- Part B propagates all necessary information along the tooth in direction r. (Remark: this is not necessary in the single-assignment version.)
- Part C implements alternative (4) of Def. 6.

$(\forall\ r\ :\ 1 \leq r \leq d\ :\ b_r^{\mathrm{pre}}(p_1, \cdots, p_r)\ :$

if $p_{r+1} = 0$ **and** $\cdots$ **and** $p_d = 0$ **then**

$A:$
/* *receive signals* */
creceive$(\mathit{chan}_{\mathrm{S}}^1, [\mathit{rcved}_1^r(p_1, \cdots, p_r)])$
$\vdots$
creceive$(\mathit{chan}_{\mathrm{S}}^{r-1}, [\mathit{rcved}_{r-1}^r(p_1, \cdots, p_r)])$

$B:$
/* *receive data* */
receive$(\mathit{chan}_{\mathrm{D}}^r, [\mathit{targ\_active}_r(p_1, \cdots, p_r),\ k_r(p_1, \cdots, p_r),$
$w_1^r(p_1, \cdots, p_r), \cdots, w_{r-1}^r(p_1, \cdots, p_r)])$

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

$C:$
$w_1^r(p_1, \cdots, p_r + 1) := w_1^r(p_1, \cdots, p_r)$ **and not** rcved_1^r
$\vdots$
$w_{r-1}^r(p_1, \cdots, p_r + 1) := w_{r-1}^r(p_1, \cdots, p_r)$ **and not** rcved_{r-1}^r

$D:$
$w_1^{r+1}(p_1, \cdots, p_r, lb_{r+1}) := \mathit{rcved}_1^r(p_1, \cdots, p_r)$
$\vdots$
$w_{r-1}^{r+1}(p_1, \cdots, p_r, lb_{r+1}) := \mathit{rcved}_{r-1}^r(p_1, \cdots, p_r)$
$w_r^{r+1}(p_1, \cdots, p_r, lb_{r+1}) := (p_r \neq lb_r)$
$\leftarrow$ only for $r < d$

$E:$ $k_r(p_1, \cdots, p_r + 1) := k_r(p_1, \cdots, p_r)$

if $\mathit{targ\_active}_r(p_1, \cdots, p_r)$ **and** $\mathit{condition}_r(p_1, \cdots, p_r)$ **then**

$F:$
$\mathit{sigval}_r(p_1, \cdots, p_r) :=$ false
$\mathit{targ\_active}_r(p_1, \cdots, p_r + 1) :=$ true
$\mathit{targ\_active}_{r+1}(p_1, \cdots, p_r, lb_{r+1}) :=$ true
$k_{r+1}(p_1, \cdots, p_r, lb_{r+1}) := r$
$\leftarrow$ only for $r < d$

else

$G:$
$w^r(p_1, \cdots, p_r) := w_1^r(p_1, \cdots, p_r + 1)$ **or** $\cdots$ **or** $w_{r-1}^r(p_1, \cdots, p_r + 1)$
$\mathit{sigval}_r(p_1, \cdots, p_r) :=$ **not** $w^r(p_1, \cdots, p_r)$
$\mathit{targ\_active}_r(p_1, \cdots, p_r + 1) :=$ false

$H:$
$\mathit{targ\_active}_{r+1}(p_1, \cdots, p_r, lb_{r+1}) :=$ false
if $\mathit{sigval}_r(p_1, \cdots, p_r)$ **then**
$k_{r+1}(p_1, \cdots, p_r, lb_{r+1}) := k_r(p_1, \cdots, p_r)$
else
$k_{r+1}(p_1, \cdots, p_r, lb_{r+1}) := r$
fi
$\leftarrow$ only for $r < d$

fi

fi

$)$

Fig. 5. The prefix of the transformed loop body

```
(∀ r : 1≤r≤d : b_r^post(p_1, ···, p_r) :

        if p_{r+1}=0 and ··· and p_d=0 then
     ⎧    if k_r(p_1, ···, p_r) = 0 and sigval_r(p_1, ···, p_r) then ⎫
 I : ⎨      Abort Server()                                            ⎬ ← only for r=d
     ⎩    fi                                                          ⎭
     ⎧   /* send data */
 J : ⎨   asend(chan_D^r, [targ_active_r(p_1, ···, p_r+1), k_r(p_1, ···, p_r+1),
     ⎩               w_1^r(p_1, ···, p_r+1), ···, w_{r-1}^r(p_1, ···, p_r+1)])
     ⎧    /* send signals */
 K : ⎨    if k_r(p_1, ···, p_r) ≠ 0 and sigval_r(p_1, ···, p_r) then
     ⎪          asend(chan_S^{k_r}, [true])
     ⎩    fi
        fi
)
```

Fig. 6. The postfix of the transformed loop body

- All but the last line of part D implements alternative (3); the last line implements alternative (2).
- Line E copies the value of k for the next iteration.
- The **if** clause after line E tests whether the current iteration must be executed with respect to level r (Def. 2) The value of $targ\_active_r$, $sigval_r$ and, if they exist, $targ\_active_{r+1}$ and k_{r+1} depend on the outcome of the test.
- sig_k^r has three conjuncts (Def. 6). The first corresponds to the **then** or the **else** branch of said **if** clause, the second to the calculation at line G. The third conjunct is satisfied by an appropriate setting of k_r in parts F and H.
- *targ_active* implements *active* (Def. 1), as proved in Lemma 14.
- Part I applies only for $r=d$. It tests for sig_0^d (i.e., $sigval_d$ and $k_d=0$) and, if so, terminates the entire program (compare Lemma 8 and Def. 5).
- Part J sends the data that are received by Part B on to the next point of the tooth.
- Part K sends sig_k^r, if it has to be sent, i.e., if *sigval* holds. (At present, we ignore signals in direction 0. In the future, we plan to use this information in the development of smarter loop bounds.)

Remark 10. In the synchronous model, we get the same semantics if the postfixes are made prefixes instead. This is an optimization if each processor has a co-processor for the transmission of messages so that computation and message handling can proceed in parallel, as is the case for the transputer [5]. For asynchronous machines, there is no similar optimization since only the receipt of messages can guarantee that the sender has updated all transformed source variables.

7.2 Correctness Proof

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

We need some additional notation. The row of $\mathcal{T}$ that represents the time component of the space-time matrix $\mathcal{T}$ shall have index t. The function $t(p)$ denotes the schedule of p.

For the following proofs, we need a formal definition of $\mathcal{TS}$, the set of target points that are scanned by the target program. Note that the target loops in Fig. 4 have no upper bounds: they enumerate an infinite set. Our way of terminating the target program is by calling the PARIX command **AbortServer** at some point. This call terminates the whole program. Thus, points of $\mathcal{TI}$ are scanned until the **AbortServer** command is issued at some point. This leads to the following definition:

Definition 11 (*targ_scanned* and $\mathcal{TS}$). The image of a point $p \in \mathcal{I}$ is scanned by the target program if all points with a time component not larger than $t(p)$ do not call the **AbortServer** command, where the call of **AbortServer** is guarded by the condition $k_r(p')=0 \wedge sigval_d(p')$ (Fig. 6, Part I). Formally:

$$(\forall\, p \,:\, p\in\mathcal{I} \,:\, \mathit{targ\_scanned}(p) =$$
$$(\forall\, p' \,:\, p'\in\mathcal{I} \wedge t(p')\leq t(p) \,:\, \neg(k_r(p')=0 \wedge \mathit{sigval}_d(p'))))$$

$$\mathcal{TS} = \{\mathcal{T}p\in\mathcal{TI} : \mathit{targ\_scanned}(p)\}$$

To be able to argue about target points whose transformed loop body is executed, we need also a formal definition of those points. The target program executes the transformed loop body iff $\mathit{targ\_active}_d(p_1,\cdots,p_d) \wedge \mathit{condition}_d(p_1,\cdots,p_d)$ (Fig. 4). Thus, we define a predicate *targ_executed* and prove (Thm. 22) that the transformed loop body is executed exactly for those points that belong to the execution space, i.e., that $\mathit{targ\_executed}(q)$ is equal to $\mathit{executed}(\mathcal{T}^{-1}q)$ for all points $q\in\mathcal{TI}$.

Definition 12 (*targ_executed$_r$* and *targ_executed*).

$$(\forall\, r \,:\, 1\leq r\leq d \,:\, (\forall\, p \,:\, p\in\mathcal{S} \,:\, \mathit{targ\_executed}_r(p_1,\cdots,p_r) =$$
$$\mathit{targ\_active}_r(p_1,\cdots,p_r) \wedge \mathit{condition}_r(p_1,\cdots,p_r)))$$

$$(\forall\, p \,:\, p\in\mathcal{S} \,:\, \mathit{targ\_executed}(p) = \mathit{targ\_executed}_d(p))$$

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

Lemma 13. *Every variable* $\mathit{targ\_active}_r(p_1,\cdots,p_r)$, $1\leq r\leq d$ *and* $(p_1,\cdots,p_r)\in\mathcal{I}$, *is assigned at most once during the execution of the transformed code.*

Proof. Every $targ\_active_r(p_1, \cdots, p_r)$ occurs exactly once as target variable of an assignment. Three disjoint cases cover every point $(p_1, \cdots, p_r)$ of $\mathcal{I}$:

1. if $(r=1 \wedge p_r = lb_r)$, it is assigned by the initialization statement $targ\_active_1(lb_1)$:= true;
2. if $(1<r<d \wedge p_r = lb_r)$, it is assigned by $targ\_active_{r+1}(p_1, \cdots, p_r, lb_{r+1})$:= true in the **then** branch of $b^{\text{pre}}_{r-1}(p_1, \cdots, p_{r-1})$;
3. if $p_r > lb_r$, it is assigned either by $targ\_active_r(p_1, \cdots, p_r+1)$:= true or by $targ\_active_r(p_1, \cdots, p_r+1)$:= false in the **then** branch or the **else** branch of $b^{\text{pre}}_r(p_1, \cdots, p_r)$, respectively.

Since no point is scanned more than once, we conclude that $targ\_active_r(p_1, \cdots, p_r)$ is assigned at most once for any $(p_1, \cdots, p_r)$.

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

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

Lemma 14. *$targ\_active_r$ implements $active_r$ for all points of $\mathcal{I}$ where the value of $targ\_active_r$ is defined, i.e., whose image is scanned by the target program. Formally:*

$$(\forall\, r, p \;:\; 1 \le r \le d \wedge \mathcal{T}p = \mathcal{T}(p_1, \cdots, p_d) \in \mathcal{TS} \;:\; targ\_active_r(p_1, \cdots, p_r) = active_r(p_1, \cdots, p_r))$$

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

- Induction base ($r = 1$):
  - Induction base ($p_1 = lb_1$):

$$\begin{array}{ll} & p_1 = lb_1 \\ \Rightarrow & \{ \text{ definition of } targ\_active \text{ and } active \text{ } \} \\ & targ\_active_1(p_1) = \text{true} = active_1(p_1) \end{array}$$

  - Induction step ($p_1 - 1 \rightarrow p_1$, where $p_1 > lb_1$):

$$\begin{array}{ll} & targ\_active_1(p_1) \\ \Leftrightarrow & \{ \text{ definition of } targ\_active_1(p_1) \text{ in the } \textbf{if} \text{ clause of } b_1(p_1-1)\text{: } \} \\ & targ\_active_1(p_1-1) \;\wedge\; condition_1(p_1-1) \\ \Leftrightarrow & \{ \text{ induction hypothesis for } p_1-1 \text{ } \} \\ & active_1(p_1-1) \;\wedge\; condition_1(p_1-1) \\ \Leftrightarrow & \{ \text{ definition of } active_r \text{ for } r=1 \text{ and } p_1 > lb_1 \text{ } \} \\ & active_1(p_1) \end{array}$$

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

$\quad targ\_active_r(p_1, \cdots, p_{r-1}, lb_r)$
$\Leftrightarrow \quad \{$ definition of $targ\_active_1(p_1)$ in the **if** clause of $b_{r-1}(p_1, \cdots, p_{r-1})$: $\}$
$\quad targ\_active_{r-1}(p_1, \cdots, p_{r-1}) \ \wedge \ condition_{r-1}(p_1, \cdots, p_{r-1})$
$\Leftrightarrow \quad \{$ induction hypothesis for $r-1$ $\}$
$\quad active_{r-1}(p_1, \cdots, p_{r-1}) \ \wedge \ condition_{r-1}(p_1, \cdots, p_{r-1})$
$\Leftrightarrow \quad \{$ definition of $active_r$ for $r>1$ and $p_r = lb_r$ $\}$
$\quad active_r(p_1, \cdots, p_r)$

- Induction step $((p_1, \cdots, p_r - 1) \to (p_1, \cdots, p_r)$, where $p_r > lb_r)$: analogously to $p_1 - 1 \to p_1$.

Next, we prove that also predicate *sig* (Def. 6) is implemented correctly. We proceed in several steps:

1. The following definition, Def. 15, gives the predicate that corresponds to *sig* in the target program a name: *TargSig*.
2. Two auxiliary lemmata lead up to Cor. 18, which expresses *TargSig* in analogy to the definition of *sig*.
3. This correspondence is helpful for the proof of Lemma 19, which states that *TargSig* implies *sig*.

According to Fig. 6, Part K, a signal is sent if $(k_r(p) \neq 0) \wedge sigval_r(p)$. The direction of this signal is $k_r(p)$. This leads to the following definition:

Definition 15 ($\boldsymbol{TargSig^r_k}$). In the target program, a signal is sent into direction k iff the direction given by $k_r(p)$ equals k and $sigval_r(p)$ holds. Formally:

$$(\forall\, r \,:\, 1 \leq r < d \,:\, (\forall\, p \,:\, \mathcal{T} p \in \mathcal{TS} \,:\, (\forall\, k \,:\, 0 \leq k < r \,:\\ TargSig^r_k(p) = (k_r(p) = k) \wedge sigval_r(p))))$$

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

Lemma 16. *If a point q is scanned by the target program, the value of $sigval_r$ (second conjunct of Def. 15) is analogous to the first two conjuncts of the definition of sig (Def. 6). Formally:*

$$(\forall\, (p_1, \cdots, p_d) \,:\, \mathcal{T}(p_1, \cdots, p_d) \in \mathcal{TS} \,:\, (\forall\, r \,:\, 0 < r \leq d \,:\ sigval_r(p_1, \cdots, p_r) \ = \\ \neg executed_r(p_1, \cdots, p_r) \wedge \\ (\forall\, s \,:\, 1 \leq s < r \,:\, \neg w^r_s(p_1, \cdots, p_r) \vee TargSig^r_s(p_1, \cdots, p_s - 1, \cdots, p_r))\ ))$$

Proof. Following the program, we distinguish two cases.

- Case 1 $(targ\_active_r(p_1, \cdots, p_r) \wedge condition_r(p_1, \cdots, p_r))$:
  In this case, $executed_r(p_1, \cdots, p_r)$ holds:

  $\quad executed_r(p_1, \cdots, p_r)$
  $\Leftrightarrow \quad \{$ Def. 2 $\}$
  $\quad active_r(p_1, \cdots, p_r) \wedge condition_r(p_1, \cdots, p_r)$
  $\Leftrightarrow \quad \{$ Lemma 14 $\}$
  $\quad targ\_active_r(p_1, \cdots, p_r) \wedge condition_r(p_1, \cdots, p_r)$
  $\Leftrightarrow \quad \{$ condition of the first case $\}$
  $\quad$ true

Thus,

$$\begin{array}{ll}
& \neg executed_r(p_1, \cdots, p_r) \;\wedge \\
& \quad (\forall\, s \,:\, 1 \leq s < r \,:\, \neg w^r_s(p_1, \cdots, p_r) \vee TargSig^r_s(p_1, \cdots, p_s - 1, \cdots, p_r)) \\
\Leftrightarrow & \{\; executed_r(p_1, \cdots, p_r), \text{ false is null of } \wedge \;\} \\
& \text{false} \\
\Leftrightarrow & \{\; \text{definition of } sigval \text{ by the program in this case } \} \\
& sigval_r
\end{array}$$

– Case 2 ($\neg(targ\_active_r(p_1, \cdots, p_r) \wedge condition_r(p_1, \cdots, p_r))$):
Analogously, in this case, we obtain $\neg executed_r(p_1, \cdots, p_r)$.Then:

$$\begin{array}{ll}
& sigval_r(p_1, \cdots, p_r) \\
\Leftrightarrow & \{\; \text{definition of } sigval \text{ in this case and definitions in Parts } C \text{ and } G \\
& \quad \text{of the program } \} \\
& \neg\big((w^r_1(p_1, \cdots, p_r) \wedge \neg rcved^r_1(p_1, \cdots, p_r)) \vee \cdots \vee \\
& \quad \neg(w^r_{r-1}(p_1, \cdots, p_r) \wedge \neg rcved^r_{r-1}(p_1, \cdots, p_r))\big) \\
\Leftrightarrow & \{\; \text{de Morgan, two times } \} \\
& (\neg w^r_1(p_1, \cdots, p_r) \vee rcved^r_1(p_1, \cdots, p_r)) \wedge \cdots \wedge \\
& \quad (\neg w^r_{r-1}(p_1, \cdots, p_r) \vee \neg rcved^r_{r-1}(p_1, \cdots, p_r)) \\
\Leftrightarrow & \{\; \text{formalization } \}
\end{array}$$

$$\begin{array}{ll}
& (\forall\, s \,:\, 1 \leq s < r \,:\, \neg w^r_s(p_1, \cdots, p_r) \vee rcved^r_s(p_1, \cdots, p_r)) \\
\Leftrightarrow & \{\; \text{definition of } rcved \;\} \\
& (\forall\, s \,:\, 1 \leq s < r \,:\, \neg w^r_s(p_1, \cdots, p_r) \vee TargSig^r_s(p_1, \cdots, p_s - 1, \cdots, p_r)) \\
\Leftrightarrow & \{\; \neg executed_r(p_1, \cdots, p_r), \text{ true is unit of } \wedge \;\} \\
& \neg executed_r(p_1, \cdots, p_r) \wedge \\
& \quad (\forall\, s \,:\, 1 \leq s < r \,:\, \neg w^r_s(p_1, \cdots, p_r) \vee TargSig^r_s(p_1, \cdots, p_s - 1, \cdots, p_r))
\end{array}$$

Lemma 17. *If point $\mathcal{T}p$ is scanned by the target program, the validity of $k_r(p_1, \cdots, p_r) = k$ (first conjunct of Def. 15) implies $TargSig^{r-1}_k(p_1, \cdots, p_{r-1})$, in analogy to the third conjunct of the definition of sig(Def. 6). Formally:*

$$\begin{array}{c}
(\forall\, r \,:\, 1 \leq r \leq d \,:\, (\forall\, p \,:\, \mathcal{T}p \in \mathcal{TS} \,:\, (\forall\, k \,:\, 0 \leq k < r \,: \\
k_r(p_1, \cdots, p_r) = k \Rightarrow (r = k+1 \;\vee\; TargSig^{r-1}_k(p_1, \cdots, p_{r-1})))))
\end{array}$$

Proof.

$$\begin{array}{ll}
& k_r(p_1, \cdots, p_r) = k \\
\Leftrightarrow & \{\; \text{definition of } k_r \text{ in the program } \} \\
& k_r(p_1, \cdots, p_{r-1}, lb_r) = k \\
\Rightarrow & \{\; \text{definition of } k_r \text{ in the program: in Fig. 4, in Part } F \text{ and in} \\
& \quad \text{the } \textbf{else} \text{ branch of part } H \; k_r \text{ is set to } r-1\text{; in the } \textbf{then} \text{ branch} \\
& \quad \text{of Part } H \text{ (where } sigval \text{ holds) it is set to } k_{r-1}(p_1, \cdots, p_{r-1}) \;\} \\
& (r-1 = k) \vee (sigval_{r-1}(p_1, \cdots, p_{r-1}) \wedge k_{r-1}(p_1, \cdots, p_{r-1}) = k) \\
\Leftrightarrow & \{\; \text{Def. 15 } \} \\
& (r = k+1) \vee TargSig^{r-1}_k(p_1, \cdots, p_{r-1})
\end{array}$$

Corollary 18. *If a point q is scanned by the target program, the conjunction of $sigval_r$ and $k_r(p_1,\cdots,p_r)=k$ (and, by Def. 15, the value of $TargSig^r_k(p)$) is analogous to all three conjuncts of the definition of sig(Def. 6). Formally:*

$$
\begin{gathered}
(\forall\, r\ :\ 1\leq r\leq d\ :\ (\forall\, p\ :\ \mathcal{T}p\in\mathcal{TS}\ :\ (\forall\, k\ :\ 0\leq k\leq d\ :\\
TargSig^r_k(p) = \neg executed_r(p_1,\cdots,p_r)\wedge\\
(\forall\, s\ :\ 1\leq s<r\ :\ \neg w^r_s(p_1,\cdots,p_r)\ \vee\ TargSig^r_s(p_1,\cdots,p_s-1,\cdots,p_r))\wedge\\
(r = k+1 \vee TargSig^{r-1}_k(p_1,\cdots,p_{r-1}))\ )))
\end{gathered}
$$

Proof.

$$
\begin{array}{ll}
 & TargSig^r_k(p)\\
\Leftrightarrow & \{\ \text{Def. 15}\ \}\\
 & sigval_r p\wedge k_r(p_1,\cdots,p_r)=k\\
\Rightarrow & \{\ \text{Lemma 16 and Lemma 17}\ \}\\
 & (\neg executed_r(p_1,\cdots,p_r)\ \wedge(\forall\, s\ :\ 1\leq s<r\ :\ \neg w^r_s(p_1,\cdots,p_r)\vee\\
 & \qquad TargSig^r_s(p_1,\cdots,p_s-1,\cdots,p_r)))\wedge\\
 & \qquad (r = k+1\ \vee\ TargSig^{r-1}_k(p_1,\cdots,p_{r-1}))
\end{array}
$$

Lemma 19 (TargSig implements sig).

$$(\forall\, r\ :\ 1\leq r\leq d\ :\ (\forall\, p\ :\ \mathcal{T}p\in\mathcal{TS}\ :\ (\forall\, k\ :\ 0\leq k<r\ :\ TargSig^r_k(p)\Rightarrow sig^r_k(p))))$$

Proof. Induction on the "distance" $N = (\Sigma\ i\ :\ 1\leq i\leq r\ :\ p_i)$ of point p from the origin

- Induction base ($N=0$):

$$
\begin{array}{ll}
 & TargSig^r_k(p_1,\cdots,p_r)\\
\Rightarrow & \{\ \text{Def. 15 (}TargSig\text{)}\ \}\\
 & (k_r(p_1,\cdots,p_r)=k)\wedge sigval_r(p_1,\cdots,p_r)\\
\Rightarrow & \{\ \text{predicate calculus}\ \}\\
 & sigval_r(p_1,\cdots,p_r)\\
\Rightarrow & \{\ \text{Lemma 16}\ \}\\
 & \neg executed_r(p_1,\cdots,p_r)\wedge\\
 & \qquad(\forall\, s\ :\ 1\leq s<r\ :\ \neg w^r_s(p_1,\cdots,p_r)\vee TargSig^r_s(p_1,\cdots,p_s-1,\cdots,p_r))\\
\Rightarrow & \{\ N=0 \text{ implies } (p_1,\cdots,p_r)=(0,\cdots,0) \text{ and}\\
 & \qquad TargSig^r_s(p_1,\cdots,p_s-1,\cdots,p_r) = \text{false} = sig^r_s(p_1,\cdots,p_s-1,\cdots,p_r)\ \}\\
 & \neg executed_r(p_1,\cdots,p_r)\wedge\\
 & \qquad(\forall\, s\ :\ 1\leq s<r\ :\ \neg w^r_s(p_1,\cdots,p_r)\vee sig^r_s(p_1,\cdots,p_s-1,\cdots,p_r))\\
\Leftrightarrow & \{\ \text{Def. 6 (}sig\text{)}\ \}\\
 & sig^r_k(p_1,\cdots,p_r)
\end{array}
$$

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

$TargSig_k^r(p_1,\cdots,p_r)$

$\Rightarrow$ { Def. 15 (*TargSig*), predicate calculus and Lemma 16, as for the induction base }

$\neg executed_r(p_1,\cdots,p_r) \wedge$
$(\forall\, s\, :\, 1 \le s < r\, :\, \neg w_s^r(p_1,\cdots,p_r) \vee TargSig_s^r(p_1,\cdots,p_s-1,\cdots,p_r))$

$\Rightarrow$ { induction hypothesis for $(p_1,\cdots,p_s-1,\cdots,p_r)$ }

$\neg executed_r(p_1,\cdots,p_r) \wedge$
$(\forall\, s\, :\, 1 \le s < r\, :\, \neg w_s^r(p_1,\cdots,p_r) \vee sig_s^r(p_1,\cdots,p_s-1,\cdots,p_r))$

$\Leftrightarrow$ { Def. 6 (*sig*) }

$sig_k^r(p_1,\cdots,p_r)$

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

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

Lemma 20. *All transformed points with inverse image in $\mathcal{X}$ are scanned if the schedule is an affine function with positive coefficients, i.e.,*

$$(\forall\, i\, :\, 1 \le i \le d\, :\, \mathcal{T}_{t,i} \ge 0) \Rightarrow (\forall\, p\, :\, p \in \mathcal{X}\, :\, \mathcal{T}p \in \mathcal{TS}).$$

Proof. Our aim is to apply Def. 11. First, we find a point $\hat{p} \in \widehat{\mathcal{X}}$ with $t(p) < t(\hat{p})$:

$p \in \mathcal{X}$

$\Rightarrow$ { Def. 1 and Def. 2 }

$active_d(p) \wedge condition_d(p)$

$\Leftrightarrow$ { Def. 1 for $(p_1,\cdots,p_r+1,\cdots,p_d)$, where $1 \le r \le d$ is a **while** dimension }

$active_r(p_1,\cdots,p_r+1,\cdots,p_d)$

$\Rightarrow$ { Def. 4 }

$(p_1,\cdots,p_r+1,\cdots,p_d) \in \widehat{\mathcal{X}}$

We name this point $\hat{p}$; $t(\hat{p}) - t(p) = \mathcal{T}_{t,r} > 0$, since r is a **while** loop.

Now, we can prove the lemma. Let p be any point in $\mathcal{X}$ and $\hat{p}$ a corresponding point in $\widehat{\mathcal{X}}$ as just defined. Applying Def. 11, we show that all points that are enumerated by time slices up to and including $t(p)$ do not abort the program:

true

$\Leftrightarrow$ { trivial quantification }

$(\forall\, p'\, :\, p' \in \mathcal{I} \wedge t(p') \le t(p)\, :\, t(p') \le t(p))$

$\Rightarrow$ { property $t(p) < t(\hat{p})$ of $\hat{p}$ and transitivity of $\le$ and $<$ }

$$
\begin{array}{ll}
 & (\forall\, p' \;:\; p' \in \mathcal{I} \wedge t(p') \leq t(p) \;:\; t(p') < t(\hat{p})) \\
\Leftrightarrow & \{\; t \text{ is an affine function } \} \\
 & (\forall p' \;:\; p' \in \mathcal{I} \wedge t(p') \leq t(p) \;: \\
 & \qquad (\Sigma\, i \;:\; 1 \leq i \leq d \;:\; \mathcal{T}_{t,i} p'_i) < (\Sigma\, i \;:\; 1 \leq i \leq d \;:\; \mathcal{T}_{t,i} \hat{p}_i)) \\
\Rightarrow & \{\; (\forall\, i \;:\; 1 \leq i \leq d \;:\; \mathcal{T}_{t,i} \geq 0) \text{ by assumption, arithmetic } \} \\
 & (\forall\, p' \;:\; p' \in \mathcal{I} \wedge t(p') \leq t(p) \;:\; (\exists\, i \;:\; 1 \leq i \leq d \;:\; p'_i < \hat{p}_i)) \\
\Leftrightarrow & \{\; \text{negation of Def. 5 } \} \\
 & (\forall\, p' \;:\; p' \in \mathcal{I} \wedge t(p') \leq t(p) \;:\; \neg m_0^d(p')) \\
\Rightarrow & \{\; \text{contrapositive of Lemma 8 } \} \\
 & (\forall\, p' \;:\; p' \in \mathcal{I} \wedge t(p') \leq t(p) \;:\; \neg sig_0^d(p')) \\
\Rightarrow & \{\; \text{Lemma 19 and Def. 15 } \} \\
 & (\forall\, p' \;:\; p' \in \mathcal{I} \wedge t(p') \leq t(p) \;:\; \neg(k_d(p') = 0) \wedge sigval_d(p')) \\
\Rightarrow & \{\; \text{Def. 11 } \} \\
 & targ\_scanned(p) \\
\Leftrightarrow & \{\; \text{Def. 11 } \} \\
 & \mathcal{T} p \in \mathcal{TS}
\end{array}
$$

Lemma 21. *targ_active ensures that all legal points are executed, i.e., that* $q \in \mathcal{TI}$ *is executed iff* $\mathcal{T}^{-1} q \in \mathcal{I}$ *is executed and* q *is scanned by the target program. Formally:*

$$(\forall\, q \;:\; q \in \mathcal{TI} \;:\; targ\_executed(q) \Leftrightarrow targ\_scanned(q) \wedge executed(\mathcal{T}^{-1} q)).$$

Proof.

$$
\begin{array}{ll}
 & targ\_executed(q) \\
\Leftrightarrow & \{\; \text{Def. 12 } (targ\_executed) \;\} \\
 & targ\_active_d(p_1, \cdots, p_d) \wedge condition_d(p_1, \cdots, p_d) \\
\Leftrightarrow & \{\; \text{Lemma 14 } \} \\
 & targ\_scanned(q) \wedge active_d(p_1, \cdots, p_d) \wedge condition_d(p_1, \cdots, p_d) \\
\Leftrightarrow & \{\; \text{Def. 2 } (executed) \;\} \\
 & targ\_scanned(q) \wedge executed(p_1, \cdots, p_d) \\
\Leftrightarrow & \{\; \text{definition of } (p_1, \cdots, p_d) \;\} \\
 & targ\_scanned(q) \wedge executed(\mathcal{T}^{-1} q)
\end{array}
$$

Theorem 22. *For affine schedules with positive coefficients, the loop body* b *is executed at* $q \in \mathcal{TI}$ *iff it is executed at* $\mathcal{T}^{-1} q \in \mathcal{I}$*. Formally:*

$$(\forall\, i \;:\; 1 \leq i \leq d \;:\; \mathcal{T}_{t,i} \geq 0) \Rightarrow (\forall\, q \;:\; q \in \mathcal{TI} \;:\; targ\_executed(q) \Leftrightarrow executed(\mathcal{T}^{-1} q))$$

Proof.

"$\Rightarrow$": part "$\Rightarrow$" of Lemma 21.
"$\Leftarrow$":

- q is scanned (Lemma 20);
- every scanned point whose inverse image is in $\mathcal{X}$ is executed (part "$\Leftarrow$" of Lemma 21).

7.3 The target program for synchronous and/or shared-memory machines

For a real implementation, we first introduce re-assignments by a simple transformation:

- all variables that are indexed with $(p_1, \cdots, p_r)$ become scalars, e.g., $\mathit{targ\_active}_r(p_1, \cdots, p_r)$ becomes $\mathit{targ\_active}_r$;
- all variables that are indexed with $(p_1, \cdots, p_r + 1)$ become renamed scalars, e.g., $\mathit{targ\_active}_r(p_1, \cdots, p_r + 1)$ becomes $\mathit{targ\_active\_out}_r$;
- all variables that are indexed with $(p_1, \cdots, p_r, lb_{r+1})$ become scalars, e.g., $\mathit{targ\_active}(p_1, \cdots, p_r, lb_{r+1})$ becomes $\mathit{targ\_active}_{r+1}$ (note the different index).

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

The more interesting implementation problem is the adaptation to the synchronous or the asynchronous model. As we mentioned already, the signaling scheme is most easily described for synchronous machines. In that case, the given target code is complete and correct.

In the asynchronous case, we can always find a space-time mapping that is scannable. (We do not prove this here.) Scannable transformations have a distinct advantage over unscannable transformations: their termination detection creates no run-time overhead. Thus, the most efficient parallel target code for an asynchronous machine is generated by a scannable transformation. Still, for whatever reasons, one has the option of an unscannable transformation also for the asynchronous model, but with a slightly modified version of the target code just presented.

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

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

if $k_r(p_1, \cdots, p_r) \neq 0$ **then**
  asend$(\mathit{chan}_{\mathrm{S}}^{k_r}, [\mathit{sigval}_r(p_1, \cdots, p_r)])$
fi

For shared-memory systems, parts B and J are superfluous. The exchange of the signals in part A and K, however, must be transformed to shared-memory access.

8 Conclusions

We are considering a loop nest that contains **while** loops and a space-time mapping $\mathcal{T}$ that parallelizes this nest. Iff $\mathcal{T}$ is unscannable in the technical sense of [6], we cannot scan the image $\mathcal{TX}$ of the execution space $\mathcal{X}$ under $\mathcal{T}$ precisely.

For this case, we have proposed a signaling scheme here with which a target loop nest can scan a finite superset $\mathcal{TS}$ of $\mathcal{TX}$ and distinguish the points of $\mathcal{TX}$ from the points of $\mathcal{TS}\setminus\mathcal{TX}$. The determination of the upper bounds for the target loops that scan $\mathcal{S}$ proceeds by neighboring communications only; after the mapping of $\mathcal{S}$ to $\mathcal{TS}$ communication distances may exceed 1 but are still constant, i.e., independent of the problem size (Explanation: two source index points with distance 1 are distinct in one coordinate, say s, and, thus, their images have as distance vector the space components of column s of $\mathcal{T}$).

The price we have to pay for unscannable space-time mappings is that the necessary scanning of a superset of $\mathcal{TX}$ increases the execution time and/or the amount of processors needed. A more detailed analysis is left to future work.

One might wonder whether unscannable space-time mappings are at all useful at such a price. It turns out that they are definitely interesting for synchronous parallelism because, in this case, there are no scannable space-time mappings for a loop nest with at least one **while** loop that is not at the outermost level.

Our approach is conservative in the sense that the target program does not execute unnecessary **while** loop iterations. Collard [3] permits the "speculative" execution of iterations that might afterwards turn out to be unnecessary. This may actually be beneficial in certain applications: Collard restricts himself to nests of **for** loops with exactly one outer **while** loop. Both his and our method achieve linear execution time, since both rest on the polytope model.

Wu and Lewis [14] separate the "loop control" part from the **while** loop body and try to determine at compile time an upper bound for the number of iterations of the **while** loop. If this is possible, the **while** loop is actually a disguised **for** loop and the usual polytope methods can be used for a parallelization. For **while** loops that do not correspond to **for** loops, they and others [13] suggest pipelining, but with methods that cannot go beyond a constant speedup.

Acknowledgements

Thanks to Jean-Franois Collard and Max Geigl for a careful reading and to Max Geigl, in addition, for numerous discussions with the first author on multi-dimensional spaces that led to the definition of *sig*.

References

1. M. Barnett and C. Lengauer. Unimodularity and the parallelization of loops. *Parallel Processing Letters*, 2(2–3):273–281, 1992.
2. J.-F. Collard. Code generation in automatic parallelizers. In C. Girault, editor, *Proc. Int. Conf. on Applications in Parallel and Distributed Computing, IFIP W.G. 10.3*, pages 185–194. North-Holland, April 1994.
3. J.-F. Collard. Space-time transformation of while-loops using speculative execution. In *Proc. 1994 Scalable High Performance Computing Conf.* IEEE Computer Society Press, May 1994.

4. E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics.* Texts and Monographs in Computer Science. Springer-Verlag, 1990.
5. I. Graham and T. King. *The Transputer Handbook.* Prentice-Hall, 1990.
6. M. Griebl and C. Lengauer. On the space-time mapping of WHILE-loops. *Parallel Processing Letters*, 1994. To appear. Also: Technical Report MIP-9304, Fakultät für Mathematik und Informatik, Universität Passau, July 1993.
7. 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.
8. 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.
9. A. Martin. The probe: An addition to communication primitives. *Information Processing Letters*, 20(3):125–130, 1985.
10. Parsytec. *PARIX 1.2 Reference Manual*, March 1993.
11. P. Quinton. The systematic design of systolic arrays. In F. F. Soulié, Y. Robert, and M. Tchuente, editors, *Automata Networks in Computer Science*, chapter 9, pages 229–260. Manchester University Press, 1987. Also: Technical Reports 193 and 216, IRISA (INRIA-Rennes), 1983.
12. S. K. Rao and T. Kailath. Regular iterative algorithms and their implementations on processor arrays. *Proc. IEEE*, 76(3):259–282, March 1988.
13. P. P. Tirumalai, M. Lee, and M. S. Schlansker. Parallelization of while loops on pipelined architectures. *J. of Supercomputing*, 5:119–136, 1991.
14. Y. Wu and T. G. Lewis. Parallelizing while loops. In H. D. Schwetman, editor, *Int. Conf. on Parallel Processing*, volume II, pages 1–8. CRC Press, 1990.
15. J. Xue. Automating non-unimodular transformations of loop nests. *Parallel Computing*, 20(5):711–728, May 1994.

This article was processed using the LaTeX macro package with LLNCS style