Parallel Processing Letters

ON THE SPACE-TIME MAPPING OF WHILE-LOOPS

MARTIN GRIEBL and CHRISTIAN LENGAUER
Fakultät für Mathematik und Informatik
Universität Passau
D–94030 Passau, Germany

Received
Revised
Accepted by

ABSTRACT
A WHILE-loop can be viewed as a FOR-loop with a dynamic upper bound. The computational model of polytopes is useful for the automatic parallelization of FOR-loops. We investigate its potential for the parallelization of WHILE-loops.

Keywords: loop parallelization, parallelizing compiler, space-time mapping.

1. WHILE-loops as FOR-loops

We denote a FOR-loop as follows:

`FOR` *index* := *lower_bound* `TO` *upper_bound* `DO` *body*

The step size (also called *stride*) of a FOR-loop is +1. (A FOR-loop with a different stride can easily be transformed to one with stride +1.) If the upper bound of the FOR-loop is smaller than the lower bound, the loop defines the empty statement.

A WHILE-loop is commonly denoted as follows:

`WHILE` *condition* `DO` *body*

One can view a WHILE-loop as a generalized FOR-loop, with a conditional upper bound that is reevaluated after every iteration:

`FOR` *new_index* := 0 `TO` (`IF` *condition* `THEN` *new_index* `ELSE` *new_index*–1) `DO` *body*

Here, *new_index* is a new variable that counts the number of iterations of the loop and that does not appear in *body*. As in normal FOR-loops, the only thing that happens to *new_index* is an implicit increment after each iteration.

With *new_index*, the upper bound of the loop is also incremented at each iteration, provided the condition holds at that point. When the condition is found to be violated, the upper bound is reduced to cause termination. We shall use the following syntax for a WHILE-loop written as a FOR-loop:

`FOR` *new_index* := 0 `WHILE` *condition* `DO` *body*

2. Mathematical Notation

Our mathematical notation follows Dijkstra [5]. 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:

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

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

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

3. The Polytope Model

3.1. FOR-loops in the polytope model

The polytope model is a useful model of computation for the static parallelization of FOR-loops. It represents the atomic iteration steps of n perfectly nested FOR-loops as the points of a polytope in $\mathbb{Z}^n$; each loop defines the extent of the polytope in one dimension. The faces of the polytope correspond to the bounds of the loops; they are all known at compile time. This enables the discovery of maximum parallelism (relative to the choices available within the method) at compile time.

In the polytope model, a FOR-loop nest is typically represented in single-assignment form by a set of recurrence equations. A recurrence equation is

$$f(p) = g(f(q_1), \ldots, f(q_k)) \qquad \text{where} \qquad p, q_1, \ldots, q_k \in \mathcal{I}$$

The polytope $\mathcal{I}$ comprises the computation points. At present, there are parallelization methods for uniform recurrences, i.e., $q_i = p + b_i$, or affine recurrences, i.e., $q_i = A_i p + b_i$ for $i = 1, \ldots, k$. Here, the b_i are constant n-vectors and the A_i are constant $n \times n$ matrices [10].

3.2. WHILE-loops in the polytope model

At compile time, a loop nest that contains one or more WHILE-loops must be modelled by a polyhedron. (A *polyhedron* can be infinite, whereas a *polytope* is finite [8].) This space of potential execution points is called the *index space* or *iteration space*; we name it $\mathcal{I}$. If the loop body has several statements that we want to consider individually, we must add one more dimension that enumerates these

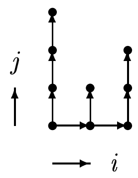

Fig. 1. A non-convex execution space with data dependences.

statements. We call the corresponding polyhedron with one more dimension the *computation space* and name it $\mathcal{C}$. The actual points of a (terminating) execution of the loop nest correspond to a finite subspace of the index space. We call this space the *execution space* and name it $\mathcal{X}$.

The WHILE-loop model departs in one important aspect from the FOR-loop model: the point that represents one loop iteration models also the boundary test preceding the iteration. One consequence is that an empty loop is modelled by one point, not by the empty set since at least its condition must be evaluated. (This is not important for the rest of this paper.)

The execution space need not be a polytope. In particular, it need not be convex.

Example

```
FOR i := 0 TO 2 DO
   FOR j := 0 WHILE condition DO
      body
```

Let the inner loop (with index j) be executed four times for $i = 0$, twice for $i = 1$, and three times for $i = 2$. This yields the non-convex execution space $\mathcal{X}$ depicted in Figure 1. Note, though, that the subspaces of $\mathcal{X}$ for every fixed i are convex, since the model of a WHILE-loop execution is a convex line. □

We can also express WHILE-loops as single-assignment recurrences. We need two equations for each WHILE-loop. One equation is the same as that for a FOR-loop; it corresponds to the iterative enumeration of the loop body, but has an unbounded index space. The other equation establishes, at every step, the validity of the condition and either defines the next iteration point or yields the result.

Example

```
y := init;
FOR i := 0 WHILE B(y) DO y := F(y);
z := y
```

where condition B and function F are without side effects.

This program corresponds to the following set of recurrence equations:

$$
\begin{array}{lcl}
y[k] & = & \texttt{IF } k=0 \texttt{ THEN } \mathit{init} \texttt{ ELSE } F(y[k-1]); \\
x[k] & = & \texttt{IF } B(y[k]) \texttt{ THEN } x[k+1] \texttt{ ELSE } y[k]; \\
z & = & x_0
\end{array}
$$

□

4. Schedules and Allocations of WHILE-loops

4.1. Schedules and allocations of FOR-loops

The *dependence graph* of a loop nest is the directed acyclic graph (V, E) whose vertex set V is the polyhedron modelling the loop nest and whose edge set E contains the direct data dependences between the computations represented by the vertices [12]. The index space, the computation space and the execution space each have an associated dependence graph.

The problem of the scheduling (in time) and allocation (in space) of polytopes modelling loops has received a lot of attention from the seminal work of 25 years ago [6] to some recent extensions [2, 9, 10].

Definition 1. (*Schedule*, *allocation*, *space-time matrix*) Let $\mathcal{I}$ be an r-dimensional polyhedron and consider the dependence graph $(\mathcal{I}, E)$.

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

  $$(\forall\ x, x' : x, x' \in \mathcal{I} \wedge (x, x') \in E : t(x) < t(x'))$$

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

- Function $a : \mathcal{I} \to \mathbb{Z}^{r-1}$ is called an *allocation* with respect to schedule t if each process it defines is internally sequential:

  $$(\forall\ x, x' : x, x' \in \mathcal{I} : t(x) = t(x') \Rightarrow a(x) \neq a(x'))$$

This is the *full-dimensional* case: space takes up $r-1$ dimensions and time the remaining one dimension of the transformed polytope (we call it the *target polytope*).

Most parallelization methods based in the polytope model require the schedule and allocation to be affine functions:

$$
\begin{array}{l}
(\exists\ \lambda, \alpha : \lambda \in \mathbb{Z}^{1\times r} \quad\ \ \wedge\ \alpha \in \mathbb{Z} \quad\ \ : (\forall\ x : x \in \mathcal{I} : t(x)\ =\ \lambda\, x + \alpha)) \\
(\exists\ \sigma, \beta : \sigma \in \mathbb{Z}^{(r-1)\times r} \wedge\ \beta \in \mathbb{Z}^{r-1} : (\forall\ x : x \in \mathcal{I} : a(x)\ =\ \sigma\, x + \beta))
\end{array}
$$

The matrix T formed by λ and σ is called a *space-time matrix*:

$$T\ =\ \left[\begin{array}{c} \lambda \\ \sigma \end{array}\right]$$

In the full-dimensional case, T is a square matrix and the requirement on the allocation is: $\mid T \mid \neq 0$. We call $T(\mathcal{I})$ (or $T(\mathcal{X})$, which will become clear from the context) the *target space*. $\square$

Lately, a relaxation to piecewise affinity has been investigated [2, 9, 10].

4.2. Schedules and allocations of one perfect loop nest including WHILE-loops

Every terminating WHILE-loop has some iteration in which some value changes such that the following iteration is disabled. This change requires a data dependence. In the static model, this dependence must be assumed between any two successive iterations of the loop. In our representation of a WHILE-loop as a generalised FOR-loop, this is made explicit by the introduction of an artificial index (Section 1). We also find this index in the according recurrence equations (Section 3.2). In this paper, we assume uniform dependences. Later work shall extend the methods to affine dependences.

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

Theorem 1. Let $v_1, \ldots, v_r$ be linearly independent vectors of $\mathbb{Z}^r$ and $\alpha_1, \ldots, \alpha_r \in \mathbb{N}\backslash\{0\}$. Then the intersection of any hyperplane H through the set of points $\{(\alpha_1\, v_1, 0, \ldots, 0), \ldots, (0, \ldots, 0, \alpha_r\, v_r)\}$ and the polyhedral cone K spanned by the vectors $v_1, \ldots, v_r$ is finite. $\square$

Proof: Our basis of $\mathbb{Z}^r$ is $\{v_1, \ldots, v_r\}$. Then $K = \{x \mid x \in \mathbb{N}^r \wedge -I\, x \leq 0\} = \mathbb{N}^r$ is the polyhedral cone spanned by $v_1, \ldots, v_r$ [8]. (I is the identity matrix.) Furthermore, $H = \{x \mid x \in \mathbb{Z}^n \wedge (\frac{1}{\alpha_1}, \ldots, \frac{1}{\alpha_r})\, x = 1\}$. Then:

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

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

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

Laying out a WHILE-loop partly in space only makes sense if we limit the number of processors required by folding the processor space in some way. This has become an active area of research recently [4, 14].

4.3 Schedules of imperfect loop nests with WHILE-loops

An imperfect loop nest specifies at least one statement that does not belong to the innermost loop of the nest. Such a statement does not belong to some (at least the innermost) loop, but succeeds it. Since the number of iterations of a WHILE-loop is not known before run time, we cannot schedule statements that succeed a WHILE-loop precisely before run time. Our solution is to enrich the schedule with additional variables—each one is a placeholder for the extent of the execution space in one dimension, i.e., for the number of actual iterations (+1) of one loop.

Example

(1) $y := init$;
(2) `FOR` $i := 0$ `WHILE` *condition* `DO` $y := F(y)$;
(3) $z := y$;

In an imperfect loop nest, the schedule and allocation must take a variable number of arguments depending on the nesting depth of the statement that is being scheduled. In this example, $t(1) = 0$, $(\forall\ i : i \in \mathbb{N} : t(2)(i) = i+1$ and $t(3) = t(2)(0) + \delta + 1 = \delta + 2$. For the loop statement (2), the schedule defines a sequence of numbers—one for each loop iteration (including the final test that leads to the termination of the loop). The schedule for the statement succeeding the loop includes the placeholder δ for the actual extent of the loop. The value of δ is not known before run time. □

4.4. Optimality

If we use affine schedules $t(i) = \lambda_1 i_1 + \cdots + \lambda_r i_r + \alpha$, for any $i = (i_1, \cdots, i_r) \in \mathcal{I}$, the coefficients $\lambda_1, \cdots, \lambda_r$ are derived from the system of linear inequalities that correspond to the dependences. If this system has only one vertex—which is frequently the case—then the optimal solution can be given at compile time, independently of the number of iterations of any WHILE-loop. If there are several vertices, the shape of the execution space influences the optimal choice of the coefficients—as is the case for a nest of FOR-loops. Therefore, the optimal choice of the parameters cannot be made before run time if WHILE-loops are considered.

5. The Target Space

Our objective is to describe the target space with a nest of loops. The image of the execution space of a loop nest may be non-convex, since even the execution space

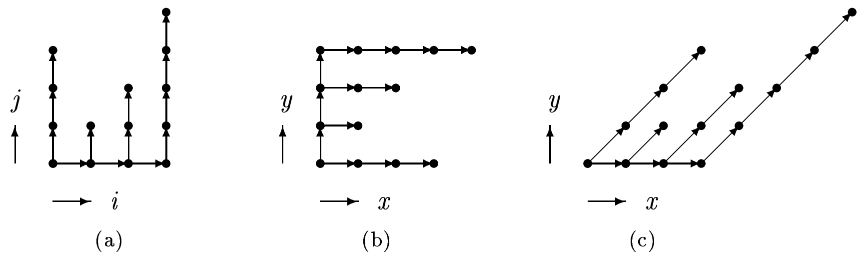

Fig. 2. Unscanable transformed execution spaces with data dependences.

itself may be. Therefore, we must address the problem of describing non-convex sets by loop nests.

5.1. Scanability

We illustrate with an example that not every transformed execution space can be scanned precisely with a nest of loops. We also state a requirement on the space-time matrix that makes the target space precisely scanable. The interpretation of an axis as being laid out in time or space does not matter.

We use the following conventions:

- We refer to the loop at level l in a loop nest as *loop* l.
- The columns of the space-time matrix T are ordered (left to right) according to the (outside-in) order of the source loop nest.
- The rows of T are ordered (top to bottom) according to the (outside-in) order of the target loop nest that we want to generate.
- A column which corresponds to a WHILE-loop is called a *WHILE-column*; the predicate $whilecol(c)$ indicates whether column c is a WHILE-column.

Example

```
FOR i := 0 TO 3 DO
    FOR j := 0 WHILE condition DO
        body
```

Figure 2(a) shows one possible execution space, Figures 2(b) and 2(c) show the transformations of this execution space by $T_1 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ and $T_2 = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$. Consider the line $x = 3$ in Figures 2(b) and 2(c). Both T_1 and T_2 define holes in this line. The distribution of these holes depends on the upper bound of the WHILE-loop which, in turn, depends on the FOR-loop index and is only known at run time. □

In order to minimize run-time overhead, we are interested in identifying the class of space-time mappings which permit a precise scanning of the target space without run-time testing for holes.* We call this property of a space-time mapping *scanability*. Our definition of scanability does not distinguish which dimensions of the space-time matrix belong to the schedule and which to the allocation. Let us first motivate it informally.

The number of iterations that are executed by a WHILE-loop w in the source loop nest L is allowed to depend on all indices of enclosing loops. All outer indices of w are known at the start of the WHILE-loop's execution. In order to obtain a loop nest L' that scans any possible transformed execution space of L precisely, we must require that:

- The terminating condition of WHILE-loop w in L might depend on the indices $c_1, \ldots, c_{w-1}$ of the enclosing loops; thus, if this WHILE-loop is transformed by space-time matrix T into a target loop w' ($T_{w',w} \neq 0$),† the same indices $c_1, \ldots, c_{w-1}$ must be derivable again—but now expressed in the indices $r_1, \ldots, r_{w'-1}$ of the target loops that enclose loop w'. We name this expression f.

- Indices of outer loops are not allowed to depend on indices of inner loops—not only in the source but also in the target loop nest. Consequently, f must not depend on indices of target loops inside loop w':

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

This leads to the following formal definition.

Definition 2. (*Scanability*) The transformation of a loop nest L by an invertible square matrix T of rank d is *scanable* iff:

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

Not surprisingly, f is part of the inverse space-time matrix T^{-1}. The following theorem states the precise definition of f.

Theorem 2. The transformation of a loop nest L by an invertible square matrix T of rank d is scanable iff:

$$\begin{array}{l}(\forall\ w, w' : 1 \leq w, w' \leq d \wedge whilecol(w) \wedge T_{w',w} \neq 0 : \\ \qquad (\forall\ r, c : 1 \leq r < w \wedge w' \leq c \leq d : T^{-1}_{r,c} = 0) \wedge w \leq w')\end{array}$$

*Holes may also arise in the space-time mapping of FOR-loops but only in regular distribution. This problem can be solved at compile time [1, 7, 11, 16].

†Note that one source loop may become part of several target loops.

Proof.

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

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

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

It follows that:

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

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

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

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

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

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

"$\Leftarrow$": Let the column w be a WHILE-column, and let $w \leq w'$ with $T_{w',w} \neq 0$. Then, let r, r', c be vectors in $\mathbb{Z}^d$ such that $r = T\,c$ and $(\forall\, i : 1 \leq i \leq w'-1 : r_i = r'_i)$. Define $f(x) = T^{-1}\big|_{1,\cdots,w-1}\, x$. We show that this choice for f satisfies the conditions required in the definition of scanability. The right side of the if-and-only-if in Theorem 2 yields:

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

□

Theorem 2 provides us with a simple way of checking whether the target space of the transformation can be scanned precisely by a target loop nest.

5.2. Choices of space-time mapping

Our requirements for precise scanning limit the choice of space-time mapping significantly. Let us illustrate what freedom of choice is left.

1. If only the outermost loop of the nest is a WHILE-loop, then every space-time mapping produces only scanable execution spaces, since the scanability condition is trivially satisfied ($1 \leq r < w$ is impossible for $w = 1$).
2. In a two-dimensional nest with an inner WHILE-loop, the inverse of the space-time matrix must have the form $\begin{bmatrix} x & 0 \\ y & z \end{bmatrix}$ with $x, y, z \in \mathbb{Z}$.
3. For deeper loop nests, there is a wide choice of space-time mappings. The following example illustrates that our definition of scanability allows for target loops with tricky bounds. Assume a nest of three loops of which only the second is a WHILE-loop, and assume the space-time mapping:

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

This target WHILE-loop does not terminate at the point when the transformed termination condition becomes invalid but a constant number of iterations later—depending on the bounds of the inner source loop. This fact also influences the bounds of the inner target loop.

In future work, we shall characterize the set of scanable target spaces further and investigate methods of code generation for target spaces that do not satisfy our definition of scanability. One technique that could be used is the synthesis of control signals [13, 15, 17].

6. Conclusions

We have shown that the parallelization method for a perfect nest of FOR-loops based on the polytope model can be extended such that perfect nests of FOR- and WHILE-loops can be mapped into space and time by using an artificial index for every WHILE-loop. Principally, every form of WHILE-loop can be handled that way.

The main complication, the introduction of a non-convex execution space, causes problems with the precise scanning of target spaces. We have formulated a requirement on the space-time matrix that is, without a program-specific data dependence analysis, necessary and sufficient for the precise scanning of target spaces. A program-specific data dependence analysis may reveal a lack of dependences that are assumed in our definition of scanability. We suspect that this may lead to more parallelism.

A more permissive approach to WHILE-loop parallelization is being developed elsewhere [3]. It permits the execution of the loop body even at some points that are not in the execution space. The resulting change of program behaviour need not always be undesirable. For example, in iterative approximations, it may lead to a higher degree of accuracy.

Acknowledgements

We thank the participants of the Dagstuhl Seminar 9325 on "Parallelization Techniques for Uniform Algorithms" for discussions at the seminar. In particular, Mike Barnett, Alain Darte and Patrice Quinton helped shape the definition of scanability. We also thank Hervé Le Verge and J.-F. Collard for very useful discussions on loop parallelization with the first author.

References

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

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

3. J.-F. Collard Space-time transformation of WHILE-loops using speculative execution, Technical Report 93-38, LIP, Ecole Nationale Superieure de Lyon, November 1993.

4. A. Darte, Regular partitioning for synthesizing fixed-size systolic arrays, *INTEGRATION* **12**, 3 (Dec. 1991) 293–304.

5. E. W. Dijkstra and C. S. Scholten, *Predicate Calculus and Program Semantics*, Texts and Monographs in Computer Science (Springer-Verlag, 1990).

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

7. W. Li and K. Pingali, A singular loop transformation framework based on non-singular matrices, Technical Report TR 92-1924, Department of Computer Science, Cornell University, July 1992.

8. G. L. Nemhauser and L. A. Wolsey, *Integer and Combinatorial Optimization*, Interscience Series in Discrete Mathematics and Optimization (John Wiley & Sons, 1988).

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

10. S. V. Rajopadhye, Synthesizing systolic arrays with control signals from recurrence equations, *Distributed Computing* **3** (1989) 88–105.

11. J. Ramanujam, Non-unimodular transformations of nested loops, in *Proc. Supercomputing '92* (IEEE Computer Society Press, 1992) 214–223.

12. S. K. Rao and T. Kailath, Regular iterative algorithms and their implementations on processor arrays, *Proc. IEEE* **76**, 3 (Mar. 1988) 259–282.

13. J. Teich and L. Thiele, Control generation in the design of processor arrays, *J. VLSI Signal Processing* **3**, 1–2 (June 1991) 77–92

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

15. J. Xue, Specifying control signals for systolic arrays by uniform recurrence equations, *Parallel Processing Letters* **1**, 2 (1992) 83–93.

16. J. Xue, An algorithm to automate non-unimodular transformations of loop nests. In *Proc. 5th IEEE Symp. on Parallel and Distributed Processing (SPDP '93)* (IEEE Computer Society Press, 1993) 512–519.

17. J. Xue and C. Lengauer, The synthesis of control signals for one-dimensional systolic arrays, *INTEGRATION* **14**, 1 (1992) 1–32.