# Exact Partitioning of Affine Dependence Algorithms

Jürgen Teich[1]* and Lothar Thiele[2]

[1] University of Paderborn, D-33100 Paderborn, Germany,
teich@date.upb.de, http://www-date.upb.de
[2] ETH Zürich, CH-8092 Zürich, Switzerland,
thiele@tik.ee.ethz.ch, http://www.tik.ee.ethz.ch

**Abstract.** This paper presents a first approach to *partition affine dependence algorithms* (e.g., sets of affine recurrence equations, loop programs with affine data dependences) when mapped onto reduced/fixed size processor arrays with local interconnect. Existing approaches start with localized dependence algorithms (e.g., sets of uniform reccurence equations, uniform loop programs). These give up optimality with respect to 1) freedom of scheduling processor tiles, 2) memory requirements due to copy operations introduced by localization of data dependences prior to partitioning, and 3) they cause unnecessary control overhead due to intermediate statements. Our partitioning approach is able to partition affine dependence algorithms and therefore represents a substantial extension of previous partitioning approaches such as described in [10, 14, 15, 11]. 4) We also propose the concept and methodology of *partial localization* that only localizes intra-tile or inter-tile data dependences, respectively, for the partitioned affine dependence algorithm.

## 1 Introduction

This paper deals with techniques for partitioning affine dependence algorithms (e.g., sets of affine recurrence equations, loop programs) onto processor arrays, e.g., VLSI processor arrays or distributed memory multiprocessors. These architectures are distinguished by the properties of distributed memory and computing power, simple and regular communication, locality in time and space, and intensive pipelining and parallelism. They are considered the most promising architectures for computationally intensive applications.

In the domain of *systolic array* processing, a typical design flow is shown in Fig. 1. Starting with a parallelized loop program with affine data dependences, or directly with a set of affine recurrence equations (AREs), data dependences are localized first. This step converts global into local data dependences. These result in local, nearest-neighbor interprocessor communications as imposed by a given interconnection topology or as a result of the mapping of operations in space and time. Transformation techniques like [13], [2] are applied that convert affine

* This work was supported by the Deutsche Forschungsgemeinschaft (DFG), SFB 376.

dependences between variables into local data propagations, see, e.g., in Fig. 2b) for the simple affine recurrence equation $x[i,j] = y[0,0]$ for all $i,j : 0 \leq i,j, < N$. The result is a (piecewise) regular algorithm (PRA) [17] or a set of uniform recurrence equations [8] (URE).

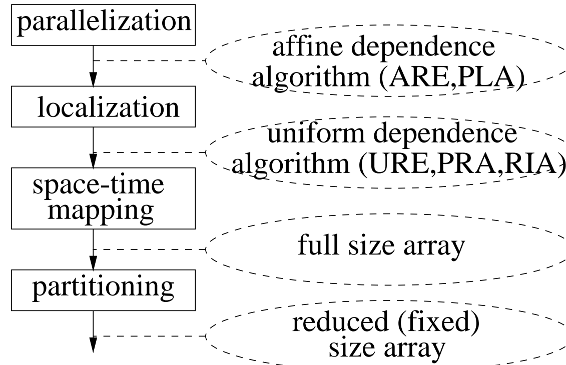


**Fig. 1.** Conventional design flow for mapping algorithms to processor arrays

After localization, linear [9] or piecewise linear [17] scheduling and allocation techniques are applied to derive a full size array. Finally, in order to match resource constraints such as limited number of processing elements, partitioning techniques have been investigated. Most of these have in common that the index space of computations is tiled using hyperplane [18], boxes [4], or parallelepiped shapes [6], [14], [1], [11]. Techniques have been proposed to either sequentially execute tiles (LPGS - local parallel, global sequential partitioning) or to sequentialize operations within a tile (LSGP - local sequential, global parallel) [7], or intermediate schemes [14]. In [3], Eckhardt and Merker propose a technique called co-partitioning that is basically a combination of LPGS and LSGP techniques for balancing memory and I/O-rate requirements while maintaining problem size independence.

The main motivation and idea of the hereafter presented *affine partitioning* solution will be explained using the following simple introductory example. In Fig. 2b), it can be seen that in case of a) where each tile is replaced by a physical processor (LSGP-scheme, see d) for the resulting array), localization of data dependences within tiles would introduce unnecessary copy operations as operations within the tile are by definition executed sequentially on the reduced size array. Also, the order of sequentially executing the operations within a tile would be unnecessarily restricted when partitioning the localized algorithm shown in Fig. 2b). Indeed, dependences between tiles need not to be localized because the physical array has the form of one tile (see d)). Localization before partitioning would unnecessarily reduce the order and amount of overlap, different tiles may be executed on the reduced size array.

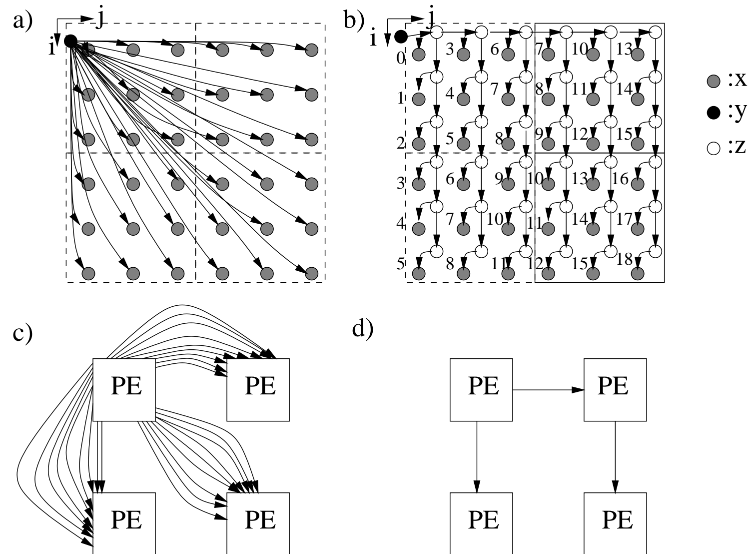


**Fig. 2.** Dependence graph of the simple affine recurrence equation $x[i,j] = y[0,0]$ and of an equivalent regular algorithm obtained by localization, tiling, and annotated schedule for LSGP partitioning. The nodes correspond to the variables $x$, $y$, and an intermediate propagation variable $z$ caused by localization, respectively. The arcs denote data dependences. In case of LSGP partitioning, each tile corresponds to one physical processor that executes its operations sequentially. c) shows the resulting processor array for the algorithm in a), d) shows the resulting processor array for the localized regular algorithm shown in b).

Localization prior to partitioning is thus inefficient in the sense that 1) many unnecessary copy operations are introduced and 2) the optimality of remaining schedules after partitioning may be greatly restricted. In the following examples, we quantify these inefficiencies concerning latency and memory overhead resulting from previous partitioning approaches for regular (localized) dependence algorithms.[1]

*Example 1.* Consider the dependence graph of the regular algorithm in Fig. 2b) that is the localized version of the affine dependence algorithm $x[i,j] = y[0,0]$ for all $i,j : 0 \leq i,j, < N$ shown in Fig. 2a). The transformed algorithm looks as follows: $x[i,j] = z[i,j]$ for all $0 \leq i,j < N$, $z[i,j] = z[i,j-1]$ for all $i = 0, 1 \leq j < N$, $z[i,j] = z[i-1,j]$ for all $1 \leq i < N, 0 \leq j < N$, and $z[i,j] = y[0,0]$ for $i = j = 0$. Note that the value of $y[0,0]$ is propagated first to each $j$ at

[1] The examples introduced throughout Section 1 to 4 are on purpose chosen to be as simple as possible in order to be able to understand the problem and to quantify the advantages of our methodology to partition affine dependence algorithms. A complex case study is given in Section 5.

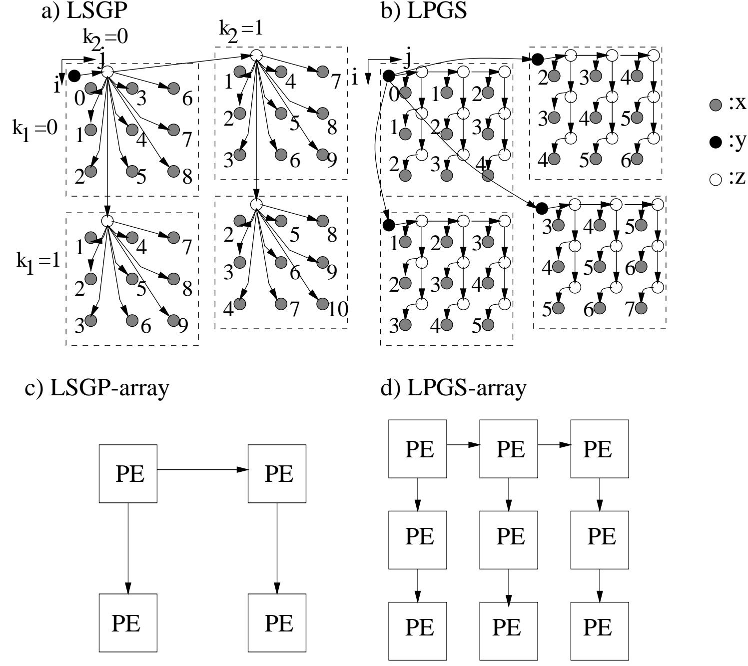


**Fig. 3.** Dependence graph of the simple affine recurrence equation $x[i,j] = y[0,0]$ during partitioning using new affine partitioning methodology: a) LSGP partitioning, b) LPGS partitititioning; included: annotated schedule. In case of LPGS partitioning, each index point within one tile corresponds to one physical processor that executes the index points at the same position of other tiles sequentially. Only dependences resulting in inter-processor communication should be localized, i.e., inter-tile data dependences in case of LSGP, and intra-tile-dependences in case of LPGS partitioning. The resulting interconnections are shown in c) for the LSGP mapping, and in d) for the LPGS-mapping.

index points $i = 0$ using the propagation direction $\begin{pmatrix} 0 & 1 \end{pmatrix}^{\mathrm{T}}$ and from there in the direction $\begin{pmatrix} 1 & 0 \end{pmatrix}^{\mathrm{T}}$ by introducing an intermediate variable $z$. Now, assume this graph is tiled using tiles of square size $M$ by $M$ as shown in Fig. 2) for $M = 3$. In case a) we decide to replace each tile by a physical processor that executes each operation within a tile sequentially (LSGP-scheme), resulting in a reduced size array of size $\lceil \frac{N}{M} \rceil \times \lceil \frac{N}{M} \rceil$, we can easily see that localization prior to partitioning introduces $M \times M$ unnessary copy operations of $y[0,0]$ (white nodes in Fig. 2b)). Hence, the local memory overhead is $\mathcal{O}(M^2)$ with respect to the solution proposed in Fig. 3a) where only one copy of $y[0,0]$ must be stored within each tile (white node). Lets look at the freedom for scheduling tiles: It can be seen in Fig. 2b) that the first time step a neighbor processor (tile $\begin{pmatrix} 1 & 0 \end{pmatrix}^{\mathrm{T}}$ or tile $\begin{pmatrix} 0 & 1 \end{pmatrix}^{\mathrm{T}}$) may start executing its first operation is at least $M$ time steps after the

first time step of operation of tile processor $\begin{pmatrix} 0 & 0 \end{pmatrix}^{\mathrm{T}}$. As can be seen in out solution in Fig. 3a), such a schedule overhead may be greatly avoided if localization is not done prior to partitioning: Obviously, only the dependences between tiles must be localized (see Fig. 3a)) allowing considerably faster schedules.

*Example 2.* In case of LPGS partitioning, the reduced size array has $M \times M$ processor elements. Correspondingly, the tiles have to executed sequentially or in a pipelined mode. Again, deficiencies in terms of redundant copy operations and restrictions in the freedom of scheduling result from localization prior to partitioning: As can be seen in Fig. 2b), the physical processor at index $\begin{pmatrix} 0 & 0 \end{pmatrix}^{\mathrm{T}}$ may start executing an operation at index points $\begin{pmatrix} kM & lM \end{pmatrix}^{\mathrm{T}}$, $k, l \in \mathbf{N}$ not before time step $kM$, $lM$, respectively. In Fig. 3b), our solution is shown that avoids this inefficiency by localization of intra-tile dependences only, however not across tiles.

A major drawback of the design flow shown in Fig. 1 is therefore that localization prior to partitioning leads to unoptimal results with respect to the latency of schedules of the reduced size array, and introduces unnecessary copy operations (additional memory requirements). In order to overcome these major drawbacks, we present to our knowledge the first systematic partitioning methodology for affine dependence algorithms. Resulting is the main message of this paper is to apply this affine partitioning methodology prior to localizing data dependences, see Fig. 4. The main results presented in this paper are summarized as follows.

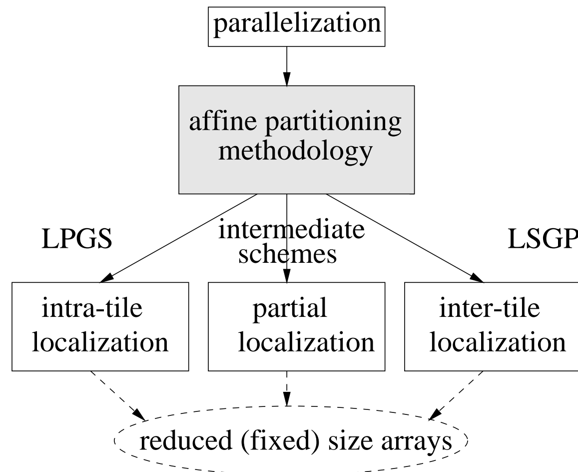


**Fig. 4.** Affine dependence algorithm partitioning methodology

– *Exact*[2] *partitioning of affine dependence algorithms* by definition of two transformations called EXPAND and REDUCE.
– *Partial localization*: By a small adaption of the EXPAND transformation, i.e., a splitting of variable transformation, only equations with inter-tile dependences must be localized subsequently in case of LS-partitioning schemes. Or, only equations with intra-tile dependences must be localized in case of a GS-partitioning scheme.
– *Hierarchical partitioning*: Finally, we will show that intermediate and hierarchical partitioning, see e.g., [4] becomes possible as special cases of how the parameters of the above transformations are chosen. This includes recent extensions to cover LPGS and LSGP partitioning schemes simultaneously such as Eckhardts [3] co-partitioning methodology.

In Section 2, we introduce the considered class of algorithms. In Section 3, the partitioning transformations EXPAND and REDUCED are introduced. It is also shown how these transformations may be used to avoid subsequent localization of intra-tile, inter-tile dependences, respectively in Section 4 (partial localization). Experimental results are reported in Section 5 for a real (FIR) digital filter design.

## 2 Regular and Affine Dependence Algorithms

In order to explain the background of our optimization methodology, we review the definitions of regular algorithms and of partitioning.

### 2.1 Regular and Affine Dependence Algorithms

The class of algorithms we are using to explain our partitioning methodology is given by extensions of the class of *Regular Iterative Algorithms* [12] which are similar to the class of *Piecewise Regular Algorithms* (PRAs), (see [17], [14]), however, with a more complex index domain, namely a *linearly bounded lattice* [16] will be named *Regular (Dependence) Algorithms* here. In case data dependences are extended to be affine, we speak of *Affine (Dependence) Algorithms*.

**Definition 1 (Affine/Regular Algorithm).** *An affine/regular algorithm consists of a set of* $|V|$ *quantified indexed equations* $S_l[I]$

$$S_1[I] \ \cdots \ S_l[I] \ \cdots \ S_{|V|}[I] \qquad \forall \ I \in \mathbf{I} \tag{1}$$

*where* $I \in \mathbf{Z}^n$ *is the index vector and the index domain* $\mathbf{I} \subseteq \mathbf{Z}^n$ *is a linearly bounded lattice that is defined below.*
*An algorithm is called* affine *if each equation* $S_l[I]$ *is of the form*

$$x_j[P_l I + f_l] = \mathcal{F}_j(\cdots x_i[Q_{ij} I - d_{ij}] \cdots) \quad \text{if } I \in \mathbf{I}_l \tag{2}$$

[2] Exact means that the transformed algorithm has the same number of computations as the original algorithm even in the existence of non-perfect tilings and that the input/output behavior of the transformed algorithm stays the same.

*where* $x_i[I]$ *are indexed variables,* $\mathcal{F}_i$ *are arbitrary functions and* $P_l, Q_{ij}, f_l, d_{ij}$ *are constant matrices and vectors of appropriate dimensions, respectively.*
*An algorithm is called* regular *if each equation* $S_i[I]$ *is of the form*

$$x_j[I] = \mathcal{F}_j(\cdots x_i[I - d_{ij}] \cdots) \quad \text{if } I \in \mathbf{I}_l \tag{3}$$

*where* $d_{ij} \in \mathbf{Z}^n$ *are constant dependence vectors. In Eq. (3) and Eq. (2),* $\mathbf{I}_l$ *is a linearly bounded lattice called* condition space *of equation* $S_l[I]$.

**Definition 2 (Linearly Bounded Lattice).** *A linearly bounded lattice is an index space of the form* $\mathbf{I} = \{I : \; I = L\kappa + m \;\wedge\; A\kappa \geq b \;\wedge\; \kappa \in \mathbf{Z}^l\}$ *where* $L \in \mathbf{Z}^{n \times l}$, $m \in \mathbf{Z}^n$, $A \in \mathbf{Z}^{m \times l}$ *and* $b \in \mathbf{Z}^m$.

$\{\kappa : \; A\kappa \geq b \wedge \kappa \in \mathbf{Z}^l\}$ defines the set of all integer vectors within a polytope. The integral points in this polytope are mapped to index vectors $I$ using an affine function, i.e., $I = L\kappa + m$. For reasons of simplicity of notation, we assume $L$ and $m$ in the description of the index space $\mathbf{I}$ in Eq. (1) to be the unity matrix and the zero vector respectively,[3] i.e., the index space can be written as $\mathbf{I} = \{I \in \mathbf{Z}^s \; : \; AI \geq b\}$.

## 3 Partitioning of Affine Dependence Algorithms

The partitioning procedure consists of two algorithm transformations called EXPAND and REDUCE. Whereas the first transformation describes the tiling and embedding of tiles in a $2n$-dimensional space, the REDUCE transformation is an affine transformation that describes a multiprojection that is able to realize all well-known and combinations of well-known partitioning schemes such as LPGS, LSGP. Later, we will also show how co-partitioning [3] can be realized.

**EXPAND-transformation:** First, the given global iteration space $\mathbf{I}$ is decomposed into the direct sum of two subspaces $\mathbf{J}$ and $\mathbf{K}$ such that $\mathbf{I} \subseteq \mathbf{J} \oplus \mathbf{K}$. The subspace $\mathbf{J} \in \mathbf{Z}^n$ represents one of the tiles and the subspace $\mathbf{K} \in \mathbf{Z}^n$ accounts for their regular repetition. Here, we consider tiles $\mathbf{J}$ that are parallelepipeds. These may be described by $n$ linearly independent tiling directions $p_i \in \mathbf{Z}^n$, $i = 1, \cdots, n$, combined in a tiling matrix $P = (p_1, \cdots, p_n)$, and a tiling offset $q \in \mathbf{Z}^n$ that describes the point where the tesselation starts.

*Example 3.* An example of a tiling of an iteration space is shown in Fig. 3 for the simple affine dependence algorithm $x[i,j] = y[0,0]$ and a tiling matrix $P = \begin{pmatrix} 3 & 0 \\ 0 & 3 \end{pmatrix}$, $q = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$. The two tiling vectors $p_1 = \begin{pmatrix} 3 \\ 0 \end{pmatrix}$ and $p_2 = \begin{pmatrix} 0 \\ 3 \end{pmatrix}$ span a rectangular tile (shown surrounded by dashed lines in Fig. 3). The tiling offset $q$ denotes at which index point the tesselation starts covering the index space.

[3] This assumption does not by any means restrict the generality of the approach: I.e., an equivalence transformation called *distribution of iteration and condition spaces* [14] may be applied to distribute the lattice mapping into the condition spaces $\mathbf{I}_l$ of each of the indexed equations in Eq. (2), and Eq. (3), respectively. As we have not formulated any restrictions on these, the approach is therefore the most general.

As tiling is not different from whether regular or affine algorithms are partitioned, we refer to [14] for details on how to compute the index space of a tile $\mathbf{J}$ and the linearly bounded lattice $\mathbf{K}$ containing all origins of non-empty tiles covered by the tiling described by tiling matrix $P$ and offset $q$:

$$\mathbf{J} = \{J \in \mathbf{Z}^n : A_J J \geq b_J\} \tag{4}$$

$$\mathbf{K} = \{K : K = P\kappa + q \ \wedge \ A_K \kappa \geq b_K \ \wedge \ \kappa \in \mathbf{Z}^n\} \tag{5}$$

*Example 4.* Consider again the tiling shown in Fig. 3. With $\mathbf{I} = \{i, j : 0 \leq i, j < N\}$, and $P, q$ taken from Example 3, we obtain

$$A_J = A_K = \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix}, b_J = \begin{pmatrix} 0 \\ -2 \\ 0 \\ -2 \end{pmatrix}, b_K = \begin{pmatrix} 0 \\ \lceil N/3 \rceil - 1 \\ 0 \\ \lceil N/3 \rceil - 1 \end{pmatrix}$$

Next, all variables are *embedded* in a $2n$-dimensional index space according to the chosen tiling. In particular, the iteration vector of the given algorithm $I \in \mathbf{I} = \mathbf{J} \oplus \mathbf{K}$ [4] will be replaced by $\hat{I} = \begin{pmatrix} J \\ K \end{pmatrix}$, $J \in \mathbf{J}$, and $K \in \mathbf{K}$. This transformation must introduce new equations with new dependences in case dependences origin from different tiles as follows. This part is much more difficult as for regular algorithms because we have to preserve affine data dependences that cross tiles instead of simple, regular dependences. Without loss of generality, we assume that equations $S_l[I]$ are all given in output normalized form:

$$x[I] = \mathcal{F}(\cdots y[QI - d] \cdots) \quad \forall I \in \mathbf{I}_l$$

In order to be able to analyze dependences between tiles easily, the above equation is transformed into the set of equivalent equations

$$\begin{aligned} x[I] &= \mathcal{F}(\cdots z[I] \cdots) \quad \forall I \in \mathbf{I}_l \\ z[I] &= x[QI - d] \quad \forall I \in \mathbf{I}_l \end{aligned} \tag{6}$$

such that dependences between different tiles may only be caused by simple equations of the form of the second equation. Now, the embedding transformation replaces the above two equations by

$$\begin{aligned} \hat{x}[J, K] &= \mathcal{F}(\cdots \hat{z}[J, K] \cdots) \quad \forall I \in \mathbf{I}_l \\ \hat{z}[J, K] &= \hat{x}[QJ - d - R, QK + R] \quad \forall J + K \in \mathbf{I}_l \ \wedge \ QJ - d - R \in \mathbf{J} \end{aligned} \tag{7}$$

where an equation of the second form is generated for all different vectors $R \in \mathbf{Z}^n$ which are solutions of the following systems of equations: In the above equation, we can write $QI - d = Q(J + K) - d = (QJ - d - R) + (QK + R)$ and we must find all possible values of $R$ for which a) $QJ - d - R \in \mathbf{J}$ and b) $QK + R \in \mathbf{K}$.

[4] $\mathbf{J} \oplus \mathbf{K} = \{I = J + K : J \in \mathbf{J} \ \wedge \ K \in \mathbf{K}\}$.

From b), $K \in \mathbf{K}$ and Eq. (5), we deduce $K = P\kappa_1 + q \ \wedge \ A_K\kappa_1 \geq b_K$ and $QK + R = P\kappa_2 + q \ \wedge \ A_K\kappa_2 \geq b_K$. Hence, $R$ must satisfy:

$$R = P\kappa_2 + q - QP\kappa_1 - Qq \ \wedge A_K\kappa_{1,2} \geq b_k \tag{8}$$

From a) and $J \in \mathbf{J}$, we deduce $A_J J \geq b_J$ and $A_J(QJ - d - R) \geq b_J$. Hence, we must introduce an equation for any distinct value $R$ in Eq. (8) obtained for any solution $\kappa_1, \kappa_2$ of the constraint polytope:

$$\begin{aligned} A_J(QJ - d - P\kappa_2 - q + QP\kappa_1 + Qq) &\geq b_J \\ A_J J &\geq b_J \\ A_K\kappa_{1,2} &\geq b_K \end{aligned} \tag{9}$$

The above polytope has $3n$ variables, and one must enumerate all its integral points to find all different solutions of $R$ for each of which a different equation is to be generated.[5]

*Example 5.* Given the affine recurrence equation $x[i,j] = y[0,0]$. With the tiling as described in the previous example, and $Q = \mathbf{0}$, $d = \mathbf{0}$, we obtain the transformed recurrence equation $\hat{x}[j_1, j_2, k_1, k_2] = \hat{y}[0,0,0,0] \quad \forall J = \begin{pmatrix} j_1 \\ j_2 \end{pmatrix} \in \mathbf{J}$, $K = \begin{pmatrix} k_1 \\ k_2 \end{pmatrix} \in \mathbf{K}$ as there is only one $R$ that satisfies the system of inequalities above: First, as $Q = \mathbf{0}$, the first set of inequalities in Eq. (9) reduces to $-A_J P\kappa_2 \geq b_J$ and is independent on $J$. Next, one can see using Eq. (8) that $\kappa_1$ has no influence on $R$ and that $\kappa_2 = \begin{pmatrix} 0 \ 0 \end{pmatrix}^{\mathrm{T}}$ leads to the only solution $R = \mathbf{0}$.

**REDUCE-transformation:** Linear transformations of the index space after embedding can now be used to permute indices, normalize lattices, etc. Using these, all known partitioning schemes can be realized, see e.g., [14] for regular algorithms:

$$I' = \begin{pmatrix} p \\ t \end{pmatrix} = \begin{pmatrix} P_J \ P_K \\ \lambda_J \ \lambda_K \end{pmatrix} \tag{10}$$

where $P_J \in \mathbf{Z}^{(n-s)\times n_J}$, $P_K \in \mathbf{Z}^{(n-s)\times n_K}$, $\lambda_J \in \mathbf{Z}^{1\times n_J}$, $\lambda_K \in \mathbf{Z}^{1\times n_K}$, and $n_J + n_K = n$. The REDUCE transformation reduces the dimension of the iterations space by $s - 1$. Variable $t$ becomes the sequencing index (time) of operations, $p$ becomes the processor index. If the decomposition $I = \begin{pmatrix} J \\ K \end{pmatrix}$ is the same as in the EXPAND transformation, then $\lambda_J$ represents the schedule of operations inside a tile, $\lambda_K$ represents the schedule of the tiles.

The freedom in the REDUCE transformation according to Eq. (10) can now be used in order to realize different partitioning schemes, i.e.

[5] This can be done quite efficiently by computing the vertices of the polytope in Eq. (9), and by testing for each integral point in its rectangular hull whether it is inside or on the border of the polytope or not. Among many investigated test cases, only very few solutions were typically detected.

- Multiprojection: $s > 1$.
- LS (Local Sequential)-partitioning: The sequentialization of operations inside a tile leads to $P_J = \mathbf{0}$. In particular, the well-known LSGP-clustering scheme is realized if $P_J = \mathbf{0}$, $P_K = E$ [6], $n - s = n_K$.
- GS (Global-Sequential)-partitioning: The sequentialization of the execution of different tiles leads to $P_K = \mathbf{0}$. In particular, the well-known LPGS-clustering is realized if $P_K = \mathbf{0}$, $P_J = E$, $n - s = n_J$.

Any intermediate partitioning scheme can be realized.

*Example 6.* Consider the simple affine dependence algorithm $y[i_1, i_2] = \mathcal{F}(z[i_1, i_1])$ with $\mathbf{I} = \{i_1, i_2 : 0 \leq i_1, i_2 < N\}$. The algorithm is already in the form that embedding may be applied directly. We have $Q = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}$ and $d = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ (tiling shown in Fig. 5). Assume we want to map the algorithm onto a processor array of fixed size 2, we choose the tiling matrix to be $P = \begin{pmatrix} \lceil \frac{N}{2} \rceil & 0 \\ 0 & N \end{pmatrix}$, $q = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ and obtain the transformed algorithm $\hat{y}[j_1, j_2, k_1] = \mathcal{F}(\hat{z}[j_1, j_1, k_1])$ if $j_1 + \lceil \frac{N}{2} \rceil k_1 < N$ as $R = \mathbf{0}$ is the only solution to the set of inequalities in Eq. (9), and as $k_2 = 0$, this index may be omitted.[7] The transformed index space is $\{ \begin{pmatrix} J \\ K \end{pmatrix} \mid 0 \leq j_1 < \lceil \frac{N}{2} \rceil, 0 \leq j_2 < N, 0 \leq k_1 < 2\}$. Fig. 5b) shows the dependence graph of the embedded affine algorithm.

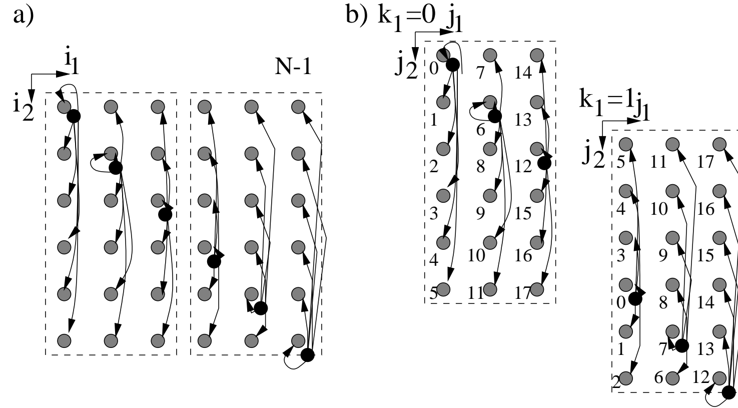


**Fig. 5.** Partitioning of an affine dependence algorithm

Finally, when applying LSGP partitioning, we see that there is no communication between the tiles. Therefore, any sequential schedule of operations within

[6] Let $E$ denote the unity matrix.
[7] Such a tiling is called *degenerated* [14].

a tile would be a feasible schedule for the transformed algorithm. We choose $P_K = 1$ and $P_J = 0$ in the corresponding REDUCE transformation. Unfortunately, there is no linear schedule $\lambda$ for executing the operations within a tile. Although any sequential schedule of operations will do (see the annotated schedule times of operations in Fig. 5b)), we might want to find a linear schedule instead. Therefore, we carry out an affine transformation of variable $\hat{z}$ replacing $\hat{z}[j_1, j_1, k_1]$ by $z[j_1, 0, k_1]$. Now, a linear schedule such as $\lambda_J = \left(j_1 + \lceil \frac{N}{2} \rceil j_2\right)$, $\lambda_K = 0$ becomes feasible.

## 4 Partial localization

In Fig. 3, we have tried to explain that in order to have local inter-processor communication in the array resulting from partitioning, the localization of data dependences is only necessary for intra-tile dependences in case of GS-partitioning and for inter-tile dependences in case of LS-partitioning schemes.

Here, we show how the embedding transformation can be modified in order to split a dependency into intra-tile dependences and inter-tile dependences, respectively. Let the embedding transformation for the second equation $z[I] = x[QI - d]$ in Eq. (6) be modified as follows:

$$\hat{z}[J, K] = \hat{x}[Q(J + K) - d, 0] \quad \forall J + K \in \mathbf{I}_l \tag{11}$$

Therefore, the indexing matrix of $\hat{x}$ becomes $Q' = \begin{pmatrix} Q & Q \\ 0 & 0 \end{pmatrix}$, and $d' = \begin{pmatrix} d \\ 0 \end{pmatrix}$. This means that $\hat{x}$ is embedded in the space $K = 0$, and therefore, no case-dependent consideration of dependences is necessary as all dependences origin from the tile at $K = 0$.

The last operation is a two-fold *splitting of inputs* transformation that transforms Eq. (11) into the equivalent system of three equations:

$$\begin{aligned} \hat{z}[J, K] &= z_1[QJ, K] \quad \forall J + K \in \mathbf{I}_l \\ z_1[QJ, K] &= z_2[QJ, QK] \quad \forall J + K \in \mathbf{I}_l \\ z_2[QJ, QK] &= \hat{x}[Q(J + K) - d, 0] \quad \forall J + K \in \mathbf{I}_l \end{aligned} \tag{12}$$

Explanations:

- The three sets of equations satisfy the single-assignment condition if variables $z_1$ and $z_2$ are names of so-far not existing variables, and as $\mathbf{N}(P_{\hat{z}}) \subseteq \mathbf{N}(Q_{z_1}) \subseteq \mathbf{N}(Q_{z_2}) \subseteq \mathbf{N}(Q_{\hat{x}})$[8].
- LS-partitioning: In case of local sequential partitioning, intra-tile dependences must not be localized. In terms of Eq. (12) and variables $\hat{z}, z_1, z_2$, and $\hat{x}$, the dependences between $z_2$ and $z_1$ (second equation) must be localized only.

[8] Let $\mathbf{N}(M)$ denote the right null space of a matrix $M$.

– GS-partitioning: Here, only intra-tile dependences must be localized. As the interconnection structure between PEs is given by the dependences caused by the first equation (between $z_1$ and $\hat{z}$), only this equation must be localized. The third equation describes copies of variables $\hat{x}$ from the border.

*Example 7.* Consider the equation $x[i,j] = y[0,0]$ with the dependence graph shown in Fig. 2a). When applying the modified embedding transformation as described in Eq. (12), we obtain the sets of equations:

$$\begin{aligned} \hat{x}[J,K] &= z[0,K] \quad \forall J+K \in \mathbf{I}_l \\ z[0,K] &= w[0,0] \quad \forall J+K \in \mathbf{I}_l \\ w[0,0] &= \hat{y}[0,0] \quad \forall J+K \in \mathbf{I}_l \end{aligned}$$

Obviously, the introduction of variable $w$ is not necessary here such that 2nd and 3rd equation may be replaced by the equation $z[0,K] = \hat{y}[0,0] \quad \forall J+K \in \mathbf{I}_l$. In case of LSGP partitioning, only this second equation must be localized. Therefore, $z[0,K] = \hat{y}[0,0]$ is replaced by $z[0,K] = z[0,K - \begin{pmatrix} 1 & 0 \end{pmatrix}^{\mathrm{T}}]$ if $k_1 > 0$, $z[0,K] = z[0,K - \begin{pmatrix} 0 & 1 \end{pmatrix}^{\mathrm{T}}]$ if $k_2 > 0 \wedge k_1 = 0$, $z[0,K] = z[0,0]$ if $K = \mathbf{0}$. The dependence graph of the resulting algorithm has already been shown in Fig. 3a).[9] A feasible REDUCE-transformation resulting in a fixed size array of $\lceil \frac{N}{M} \rceil \times \lceil \frac{N}{M} \rceil$ processing elements is given in Eq. (13).

$$\begin{pmatrix} p \\ t \end{pmatrix} = \overbrace{\begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & M & 1 & 1 \end{pmatrix}}^{LSGP} \cdot \begin{pmatrix} j_1 \\ j_2 \\ k_1 \\ k_2 \end{pmatrix} \tag{13}$$

Finally, in case of LPGS partitioning, only the first equation must be localized. Therefore, $\hat{x}[J,K] = z[0,K]$ is replaced by $\hat{x}[J,K] = \hat{x}[J - \begin{pmatrix} 1 & 0 \end{pmatrix}^{\mathrm{T}}, K]$ if $j_1 > 0$, $\hat{x}[J,K] = \hat{[}J - \begin{pmatrix} 0 & 1 \end{pmatrix}^{\mathrm{T}}, K]$ if $j_2 > 0 \wedge j_1 = 0$, $\hat{x}[0,K] = z[0,0]$ if $J = \mathbf{0}$. The dependence graph of the resulting algorithm is shown in Fig. 3b).[10] A feasible REDUCE-transformation resulting in a fixed size array of $M \times M$ processing elements is given in Eq. (14):

$$\begin{pmatrix} p \\ t \end{pmatrix} = \overbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 2 \end{pmatrix}}^{LPGS} \cdot \begin{pmatrix} j_1 \\ j_2 \\ k_1 \\ k_2 \end{pmatrix} \tag{14}$$

[9] Corresponding schedule times of operations are also annotated to the nodes in Fig. 3a).

[10] Corresponding schedule times of operations are also annotated to the nodes in Fig. 3b).

## 5 Hierarchical Partitioning and Case Study

In [3], Eckhardt and Merker introduce the term *co-partitioning* based on the idea to combine LPGS partitioning and LSGP partition by applying them one after the other with the goal to balance local memory and communication.

Using the example of an FIR-filter algorithm, we show that this hierarchical partitioning approach can be also handled as a special case of our affine partitioning methodology.

*Example 8.* The equation $y_{i_2} = \sum_{i_1=0}^{\min\{i_1,N-1\}} A_{i_1} \cdot U_{i_2-i_1}$ with $0 \leq i_2 < T$ is the difference equation of a finite impulse response (FIR)-digital filter. The functionality of this equation can be captured by the following affine algorithm:

$$y[i_1,i_2] = y[i_1-1,i_2] + a[i_1,i_2] * u[i_1,i_2] \quad (15)$$

$$a[i_1,i_2] = a[i_1,0] \quad (16)$$

$$u[i_1,i_2] = u[0,i_2-i_1] \quad (17)$$

with the index domain $\mathbf{I} = \{(i_1,i_2) : \; 0 \leq i_1 \leq N-1 \;\wedge\; 0 \leq i_2 \leq T-1\}$. Let us assume we want to partition this algorithm such that the operations are executed on a fixed size processor array with $\#PEs = 4$ processing elements (PEs).

### 5.1 Co-partitioning of affine dependence algorithms

We apply co-partitioning by performing an LPGS-tiling $P_{\mathrm{LPGS}}$, followed by a subsequent LSGP-tiling $P_{\mathrm{LSGP}}$ (see Fig. 6a)) with

$$P_{\mathrm{LPGS}} = \begin{pmatrix} N & 0 \\ 0 & x \end{pmatrix}; \quad P_{\mathrm{LSGP}} = \begin{pmatrix} N & 0 & 0 \\ 0 & y & 0 \\ 0 & 0 & z \end{pmatrix} \quad (18)$$

with $x = 8$, $y = 2$, and $z = \lceil T/x \rceil$.

The transformed algorithm obtained after these two subsequent EXPAND transformations is:

$$y[j_1,j_2,k,l] = y[j_1-1,j_2,k,l] + a[j_1,j_2,k,l] * u[j_1,j_2,k,l] \quad (19)$$

$$a[j_1,j_2,k,l] = a[j_1,0,0,0] \quad (20)$$

$$u[j_1,j_2,k,l] = u[0,j_2-j_1+4k+2l,0,0] \quad (21)$$

Obviously, the data dependences of Eq. (19) do not cause any global communication. The data dependency in Eq. (20), however, must be localized partially. $l$ denotes the processor index of the final array obtained by co-partitioning, so we see that variables at processors $l > 0$ have to read data from the processor at location $l = 0$ (border), hence the equation causes inter-processor communication.

Following our methodology of partial localization presented in Section 4, we therefore localize these data dependencies. Let $J = (j_1, j_2, k)^{\mathrm{T}}$ and $K = l$, we obtain the transformed set of equations:

$$a[j_1, j_2, k, l] = z_1[j_1, 0, 0, l] \tag{22}$$
$$z_1[j_1, 0, 0, l] = z_2[j_1, 0, 0, 0] \tag{23}$$
$$z_2[j_1, 0, 0, 0] = a[j_1, 0, 0, 0] \tag{24}$$

Obviously, we can simplify this set of equations as follows: 1) There is no need for introducing the variable $z_2$. Instead, we just delete Eq. (24) and replace $z_2$ in Eq. (23) by $a$. Now, only this 2nd equation (Eq. (23)) leads to interprocessor communication and must be localized leading to the set of transformed equations:

$$a[j_1, j_2, k, l] = z_1[j_1, 0, 0, l] \tag{25}$$
$$z_1[j_1, 0, 0, l] = z_1[j_1, 0, 0, l-1] \quad \text{if } l > 0 \tag{26}$$
$$= a[j_1, 0, 0, 0] \quad \text{if } l = 0 \tag{27}$$

A final simplification is possible here by renaming $z_1$ by $a$ such that the final result after partial localization is the set of equations:

$$y[j_1, j_2, k, l] = y[j_1 - 1, j_2, k, l] + a[j_1, j_2, k, l] * u[j_1, j_2, k, l] \tag{28}$$
$$a[j_1, j_2, k, l] = a[j_1, 0, 0, l-1] \quad \text{if } l > 0 \tag{29}$$
$$= a[j_1, 0, 0, 0] \quad \text{if } l = 0 \tag{30}$$
$$u[j_1, j_2, k, l] = u[j_1 - 1, j_2 + 1, k, l-1] \quad \text{if } j_1 > 0 \ \wedge \ l > 0 \ \wedge \ j_2 = 0 \tag{31}$$
$$= u[j_1 - 1, j_2 - 1, k, l] \quad \text{if } j_1 > 0 \ \wedge \ l \geq 0 \ \wedge \ j_2 > 0 \tag{32}$$
$$= u[j_1 - 1, j_2 + 1, k-1, l+3] \quad \text{if } j_1 > 0 \ \wedge \ l = 0 \ \wedge \ j_2 = 0 \tag{33}$$
$$= u[0, j_1 - j_2 + 4k + 2l, 0, 0] \quad \text{if} j_1 = 0 \tag{34}$$

In Eq. (31)-(34), the same idea of only partially localizing dependencies for variable $u$ has been carried out. A final REDUCE transformation leading to an array with a fixed number of $\lceil x/y \rceil = 4$ PEs ($l = 0, 1, 2, 3$) is

$$I = \begin{pmatrix} p \\ t \end{pmatrix} = \begin{pmatrix} P_J \ P_K \ P_L \\ \lambda_J \ \lambda_K \ \lambda_L \end{pmatrix}; P_L = 1, P_J = P_K = 0.$$

with the feasible schedule (see Fig. 6b)) $\lambda_J = \begin{pmatrix} y \ 1 \end{pmatrix} = \begin{pmatrix} 2 \ 1 \end{pmatrix}$, $\lambda_K = 2N = 12$, and $\lambda_L = y = 2$. The resulting processor array with 4 PEs (that sequentially computes the operations at $k = 0, k = 1, \ldots$, is shown in Fig. 7. The black boxes denote registers.

The number of registers in each inter-processor connection is obtained by multiplying the dependency vector of the corresponding dependency in Eq. (28)-Eq. (34) by the schedule vector $\lambda = (\lambda_J \ \lambda_K \ \lambda_L)$, e.g., the vector $d_{yy} = (1\ 0\ 0\ 0)$

results in a connection with $\lambda d_{yy} = (2\ 1\ 12\ 2)d_{yy} = 2$ registers in a feedback path. Similarly, the dependency $d_{aa} = (0\ 0\ 0\ 1)$ results in 2 registers. Finally, variable $u$ is either a) read in from the external or b) passed from the previous processor (Eq. (31)) where $d_{uu} = (1\ -1\ 0\ 1)$ results in a link with 3 registers, or c) read internally from a feedback interconnection (Eq. (32)) where $d_{uu} = (-1\ 1\ 0\ 0)$ results in a link with 3 registers, or d) where the processor at index $l = 0$ reads data from the processor at $l = 3$ leading to a wrap-around interconnection with $\lambda\ (-1\ 1\ 1\ -3) = 7$ registers. The control circuitry for controlling the multiplexers is not shown in Fig. 7.

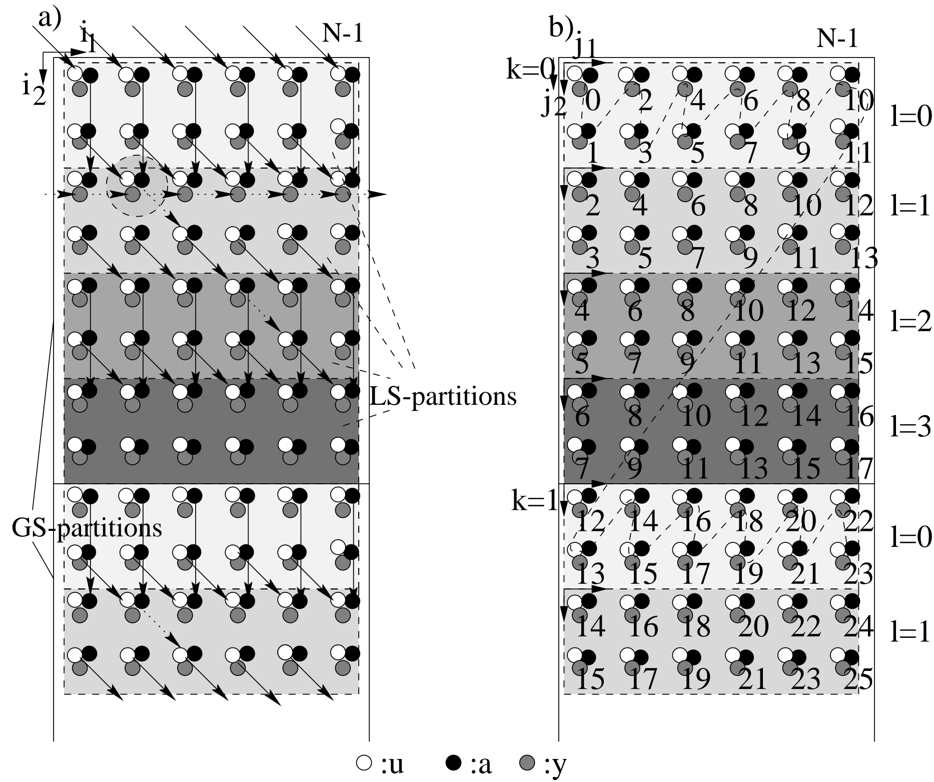


**Fig. 6.** Example of co-partitioning the depencence graph (only arcs at some index points shown) of an FIR-filter by applying two subsequent tilings after another. The first one is an LPGS-based partitioning (tiles surrounded by solid lines), the second is an LSGP-based scheme (tiles bounded by dotted lines) (a). In b), the embedded algorithm as well as a schedule is shown for each of the 4 resulting processing elements.

### 5.2 Optimization

For given $N$ and $T$, we finally would like to explore the influence of the sizes of parameterized tiling matrices in Eq. (18) during co-partitioning, i.e., of $x$ and $y$ on the number of PEs of the resulting array and on the timing, i.e., on the maximum and average computation rate of filter outputs.

The number of PEs of the resulting array, denoted $\#PEs$, is given by the number of LS-partitions:

$$\#PEs = \lceil x/y \rceil \tag{35}$$

It turns out that depending on $x$ and $y$, the schedule of the REDUCE transformation may be written also in a parameterized way:

$$\lambda = (\lambda_J \ \lambda_K \ \lambda_L) = (y \ 1 \ yN \ y)$$

Now filter outputs are computed at index points $j_1 = N - 1$. From Fig. 6, it can be seen that for the chosen values $x = 8, y = 4$ of the co-partitioning tiling matrices, the time between successive filter outputs is minimally 1 and maximally 5 clock cycles. The maximum delay is caused by the last output of one LPGS partition at index $k, l = l_{\max} = \#PEs - 1$ and the first output of a valid output at the subsequent partition $k + 1$. This happens at index $j_1 = N - 1$, $j_2 = 0$, $l = 0$. Hence, the maximum distance in time between two successive inputs, denoted $L_{\max}$ in the following, arises between scheduling of an index point $I = (N - 1 \ 0 \ k + 1 \ 0)^{\mathrm{T}}$ and point $I' = (N - 1 \ \ y - 1 \ \ k \ \ l_{\max})^{\mathrm{T}}$. Therefore, the distance in time is $\lambda(I - I') = (y \ 1 \ yN \ y)(0 \ (1 - y) \ 1 \ (-l_{\max}))^{\mathrm{T}}$ $= y(N - l_{\max} - 1) + 1$. We obtain

$$L_{\max} = y(N - \lceil x/y \rceil - 2) + 1 \tag{36}$$

The final objective, we would like to compute in dependence on the tiling is the *average latency per output* of the array, denoted $L_{\mathrm{avg}}$ in the following. Obviously, with the parameterized schedule above, the array outputs $x - 1$ outputs in successive cycles and then requires $L_{\max}$ cycles for the next sample, and so on. Hence, $L_{\mathrm{avg}}$ is given as follows:

$$L_{\mathrm{avg}} = ((x - 1)1 + L_{\max})/x \tag{37}$$

In Table 1, we report the computed objectives $\#PEs$, $L_{\max}$, and $L_{\mathrm{avg}}$ for different tile size parameters $x$ and $y$ for $N = 6$ as shown in Fig. 6. First, it can be seen from these numbers that for any fixed value of the size $x$ of the LPGS partition, $L_{\max}$ and $L_{\mathrm{avg}}$ monotonically increase for increasing values of $y$ (parameter in the LSGP partitioning matrix). Hence, for each value of $x$, a small value for $y$ gives the best latencies. However, this comes at the prices of having the largest number of PEs for small values of $y$. Secondly, it can be observed that there is an upper limit for the number of PEs that are useful in the sense to decrease the latency. With $x = 12$ and $y = 2$, we obtain the smallest array with a size of $\#PEs = 6$ that is able to output one filter output at each clock cycle.

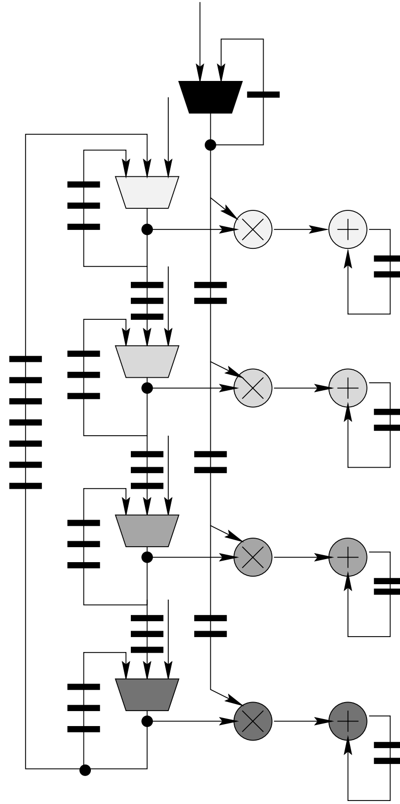

**Fig. 7.** Fixed size processor array for the FIR-filter algorithm with 4 processing elements corresponding to the 4 LS-partions shown in Fig. 6a) (control circuitry not shown)

## 6 Conclusions

In this paper, we presented a first approach to exactly partition affine dependence algorithms. With the major advantage to avoid a) unnessary copy operations created by localization prior to partitioning and b) to restrict the freedom of scheduling in early design phases of processor arrays, the message of the paper for algorithm and parallel architecture designer is to *do partitioning first!* in the design flow using the here presented transformations for tiling, embedding and partial localization. The presented transformations have been implemented for non-parameterized index spaces in the PARO design system for mapping computation intensive algorithms to hardware, see **http://www-date.upb.de/RESEARCH/PARO/**.

In the future, we would like to work out formulas for quantifying the savings obtained by our new design flow as shown in Fig. 4 in terms of latency and

| x | y | x/y | $\#PEs$ | $L_{\max}$ | $L_{\text{avg}}$ |
|---|---|---|---|---|---|
| 2 | 2 | 1 | 1 | 11 | 6.0 |
| 4 | 2 | 2 | 2 | 9 | 3.0 |
| 4 | 4 | 1 | 1 | 21 | 6.0 |
| 6 | 2 | 3 | 3 | 7 | 2.0 |
| 6 | 4 | 1.5 | 2 | 9 | 2.6 |
| 6 | 6 | 1 | 1 | 36 | 6.8 |
| 8 | 2 | 4 | 4 | 5 | 1.5 |
| 8 | 4 | 2 | 2 | 17 | 3.0 |
| 8 | 6 | 1.3 | 2 | 25 | 49.0 |
| 8 | 8 | 1 | 1 | 49 | 7.0 |
| 10 | 2 | 5 | 5 | 3 | 1.2 |
| 10 | 4 | 2.5 | 3 | 13 | 2.2 |
| 10 | 6 | 1.6 | 2 | 25 | 3.4 |
| 10 | 8 | 1.25 | 2 | 33 | 4.2 |
| 10 | 10 | 1 | 1 | 51 | 6.0 |
| 12 | 2 | 6 | 6 | 1 | 1.0 |
| 12 | 4 | 3 | 3 | 13 | 2.0 |
| 12 | 6 | 2 | 2 | 25 | 3.0 |
| 12 | 8 | 1.5 | 2 | 33 | 3.67 |
| 12 | 10 | 1.2 | 2 | 41 | 4.3 |
| 12 | 12 | 1 | 1 | 61 | 6.0 |

**Table 1.** Design space exploration of $\#PEs$, $L_{\max}$, and $L_{\text{avg}}$ in dependence of the parameters of the co-partitioning tiling matrices in Eq. (18) for the FIR-filter algorithm and problem size $N = 6$.

memory. Also, we would like to work on automatic design space exploration techniques for partitioning schemes like co-partitioning. First results on automatic exploration of space time mappings have been described in [5].

## References


1. A. Darte. Regular partitioning for synthesizing fixed size systolic arrays. *INTEGRATION, the VLSI Journal*, 14:293–304, 1991.
2. V. V. Dongen and P. Quinton. Uniformization of linear recurrence equations: A step towards the automatic synthesis of systolic arrays. In *Proc. Int'l Conf. Systolic Arrays*, pages 403–412, San Diego, 1988.
3. U. Eckhardt and R. Merker. Co-partitioning - a method for hardware/software codesign for scalable systolic arrays. In R. Hartenstein and V. Prasanna, editors, *Reconfigurable Architectures*, pages 131–138, IT Press, Chicago, IL, 1997.
4. U. Eckhardt and R. Merker. Hierarchical algorithm partitioning at system level for an improved utilization of memory structures. *IEEE Trans. on Computer-Aided Design*, 18(1):14–24, Jan. 1999.
5. Frank Hannig and Jürgen Teich. Design Space Exploration for Massively Parallel Processor Arrays. In *Sixth International Conference on Parallel Computing Technologies (PaCT-2001)*, volume 2125 of *Lecture Notes in Computer Science*, pages 51–65, Novosibirsk, Russia, September 2001. Springer-Verlag.

6. F. Irigoin and R. Triolet. Supernode partitioning. In *Proc. SIGPLAN*, pages 319–329, San Diego, Jan. 1988.
7. K. Jainandunsing. Optimal partitioning scheme for wavefront/systolic array processors. In *Proc. IEEE Symp on Circuits and Systems*, 1986.
8. R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14:563–590, 1967.
9. R. H. Kuhn. Transforming algorithms for single-stage and VLSI architectures. *Workshop Interconnection Networks for Parallel and Distributed Processing*, 1980.
10. D. I. Moldovan and R. A. B. Fortes. Partitioning and mapping of algorithms into fixed size systolic arrays. *IEEE Trans. Computers*, C-35:1–12, 1986.
11. T. P. Plaks. *Multidimensional Piecewise Regular Arrays.* PhD thesis, School of Electrical and Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, 1997.
12. S. K. Rao. *Regular iterative algorithms and their implementations on processor arrays.* PhD thesis, Stanford University, 1985.
13. V. Roychowdhury, L. Thiele, S. K. Rao, and T. Kailath. On the localization of algorithms for VLSI processor arrays. *in: VLSI Signal Processing III, IEEE Press, New York*, pages 459–470, 1989.
14. J. Teich and L. Thiele. Partitioning of processor arrays: A piecewise regular approach. *INTEGRATION: The VLSI Journal*, 14(3):297–332, 1993.
15. J. Teich, L. Thiele, and L. Zhang. Partitioning processor arrays under resource constraints. *Int. Journal of VLSI Signal Processing*, 17(1):5–20, September 1997.
16. L. Thiele. On the hierarchical design of VLSI processor arrays. In *IEEE Symp. on Circuits and Systems*, pages 2517–2520, Helsinki, 1988.
17. L. Thiele. On the design of piecewise regular processor arrays. In *Proc. IEEE Symp. on Circuits and Systems*, pages 2239–2242, Portland, 1989.
18. M. W. Wolf and M. S.Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2:452–471, 1991.