

# Hierarchical Partitioning for Piecewise Linear Algorithms

Hritam Dutta, Frank Hannig and Jürgen Teich  
Department of Computer Science 12, Hardware-Software-Co-Design,  
University of Erlangen-Nuremberg, Germany,  
{dutta, hannig, teich}@cs.fau.de

## Abstract

*processor arrays can be used as accelerators for a plenty of data flow-dominant applications. The explosive growth in research and development of massively parallel processor array architectures has lead to demand for mapping tools to realize the full potential of these architectures. Such architectures are characterized by hierarchies of parallelism and memory structures, i.e. processor array apart from different levels of Cache have large number of processing elements (PE) where each PE can further contain Sub-Word parallelism. In order to handle large scale problems, balance local memory requirements with I/O-bandwidth, and use different hierarchies of parallelism and memory one needs sophisticated transformation called hierarchical partitioning. In this paper, we introduce for the first time a detailed methodology encompassing hierarchical partitioning.*

## 1 Introduction

The perennial need for faster computation by exploiting concurrency has driven the development of parallel architectures. Such architectures are ubiquitous in modern computing systems ranging from very large instruction word (VLIW) processors to supercomputers. On one hand, general purpose parallel computers have farms of SIMD or MIMD multiple processors. These platforms offers an optimal platform for the parallel execution of number crunching algorithms from the fields of digital signal processing, image processing, numerical simulation, visualization, etc. Therefore, a major field of research is the automatic parallelization of such software loops (e.g. *for* or *while* loops in C) onto parallel architectures. The polytope model is an intuitive methodology for loop parallelization and mapping of loop nests onto parallel systems [10]. SUIF and PIPS are state-of-the-art parallelizing compiler based on polytope model [8]. On the other hand, reducing integration densities and consideration of area, cost, and power for reasons of mobility, portability has lead to development of *domain-specific* architectures. There are numerous well known software/hardware alternatives, including digital signal processors (DSPs), custom application-specific integrated circuits (ASICs), application specific instruction processors (ASIPs) including graphic processors and field programmable gate arrays (FPGAs) and also lesser known ones such as coarse-grained processor arrays (CGAs). DSPs and ASIPs exploit instruction level parallelism (ILP) which may not suffice for many applications thus calling for flexible alternatives such as multi-DSP systems, ASICs, FPGAs, etc. These solutions can exploit both *loop or iteration level parallelism*. The accelerators for loop programs in ASICs, CGAs and FPGAs can be realized as *massively parallel processor arrays*. Processor arrays consist of an array of 1-D or 2-D processing elements (PE) that may contain sub-word processing units with only very few memory and regular interconnect structures. Processor arrays are girdled by memory banks and hierarchy of caches for parallel and fast access of data. These architectures also calls for mapping tools in order to realize their full potential. The mapping tools should be able to obtain synthesizable architecture descriptions (ASICs, FPGAs) or generate program

for a given fixed processor array architecture (e.g. CGAs) from software descriptions. In last two decades lot of research in academia and industry has spawned state-of-the-art design tools as PICO-Express [13], MMAlpha [2], PARO [12], etc. A perpetual challenge of such tools is in matching resource constraints of given architecture (e.g. number of PEs, functional units, memory banks, I/O pins) with parallelized program or generated hardware description through interaction of program transformations and architecture constraints. *Partitioning* is a fundamental transformation for parallelizing compilers which tackles the problem of hardware matching. Partitioning deals with the division of the index space of the loop program into disjoint subsets called *tiles* and schedule the corresponding iterations. The popularly known partitioning schemes are local parallel global sequential (LPGS), or local sequential global parallel (LSGP) scheme. The massively parallelism might be expressed by different types of hierarchical parallelism: (1) several parallel working processing elements (PEs), (2) functional and software pipelining, (3) multiple functional units within one PE, and finally, (4) sub-word parallelism (SWP) within the PEs. The mentioned partitioning schemes do not suffice to match available massive parallelism with I/O rate and hierarchical memory constraints. Therefore *co-partitioning* which partitions an already partitioned index space was introduced with enhanced hardware matching [5]. This idea has been generalized to *n-hierarchical partitioning* where a index space is recursively tiled  $n$  times in order to match hierarchical massive parallelism, hierarchical memory constraints, and I/O rates. In this paper, we introduce for the first time a detailed methodology encompassing hierarchical partitioning.

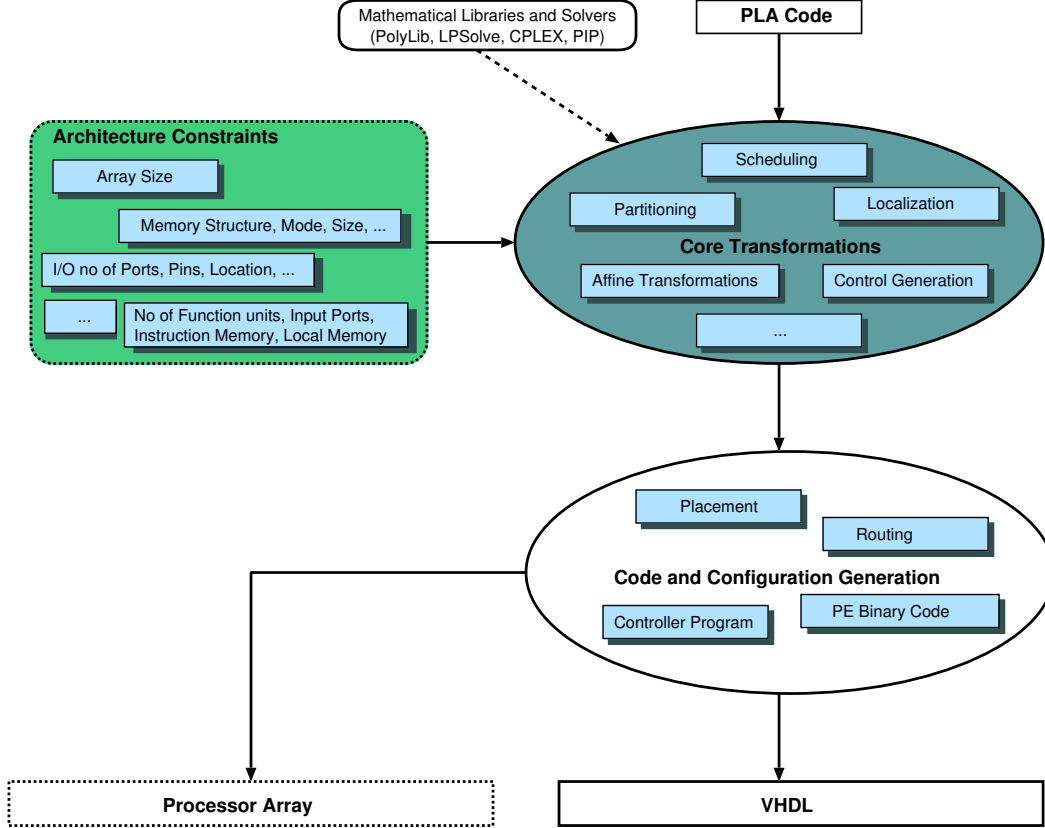
However first in section 2, we introduce the related work. Subsequently in section 3, a brief description of our design flow along with definitions and notations is given. Section 4 contains a intuitive introduction and algebraic formulation of partitioning transformation. In section 5, we unveil our exact methodology for hierarchical partitioning. Finally in section 6 we conclude and present the outlook for future work.

## 2 Related Work

Partitioning is known under modern compiler theory under different terminology like strip-mining, *loop tiling* [16]. The traditional loop tiling as in compiler theory introduces integer division, mod, ceil, and floor operators which lead to index spaces that are no convex sets. This is not only inefficient for hardware designs but also are not allowed in recurrence equations [9] which form the underlying framework of design methodologies in polytope model. The DTSE methodology [1] from IMEC is another compilation method based on the polytope model which has studied hierarchical partitioning but only for general purpose embedded processors and not processor arrays or multi-processors. PIPS tries to solve the problem of hardware matching by multi-dimensional scheduling (in other words partitioning in processor-time domain). Outstanding among all partitioning schemes is the introduction of the idea of co-partitioning in [4]. However the exact methodology for dealing the idea is not studied.

## 3 Background and Notation

Few compiler techniques for mapping loop algorithms onto massively parallel architectures are based on loop parallelization in polytope model. The theory has its origin in research for automatic generation of systolic arrays. However it was lacking in treatment of compilation of loop algorithms under architectural constraints, i.e. for a given fixed processor array architecture. Several transformations such as partitioning, localization, etc. have been proposed for loop compilation in the polytope model [14]. In this section we first give a brief overview of our existing mapping methodology PARO. The design flow of our approach is given in Fig. 1. The starting point is algorithmic description as a set of recurrence equation. The algorithm descriptions are then further transformed by *embedding* of variables, *localization*(vectorization), and other transformations in the polytope model for reasons of hardware generation. Furthermore a space-time mapping of the transformed program is carried out in order to obtain a architecture description for input to a back-end code generator for compiling the program onto a given processor array architecture. For the sake of brevity, we refer to [6] for an introduction to space-time mapping and the back-end code generator. First we recapitulate the class of algorithms we are dealing with called *piecewise*



**Figure 1. PARO Design Flow.** Our current design flow for developing VLSI processor array architectures is determined by the transformations in solid ellipses. Here, we extend the design flow (dashed boxes) to handle fixed array architectures by introduction of architectural constraints and parameter matching.

linear algorithms (PLAs).

**Definition 3.1** (PLA). A piecewise linear algorithm consists of a set of  $N$  quantified equations,  $S_1 [I], \dots, S_i [I], \dots, S_N [I]$ . Where each equation  $S_i [I]$  is of the form

$$\forall I \in \mathcal{I}_i : x_i [P_i I + f_i] = \mathcal{F}_i (\dots, x_j [Q_j I - d_{ji}], \dots) \quad \text{if } \mathcal{C}_i^1(I) \quad (1)$$

where  $x_i, x_j$  are linearly indexed variables,  $\mathcal{F}_i$  denote arbitrary functions,  $P_i, Q_j$  are constant rational indexing matrices and  $f_i, d_{ji}$  are constant rational vectors of corresponding dimension. The dots  $\dots$  denote similar arguments.  $I \in \mathcal{I}_i \subseteq \mathbb{Z}^n$  is a linearly bounded lattice (definition follows), called iteration space of the quantified equation  $S_i [I]$ . The set of all vectors  $P_i I + f_i, I \in \mathcal{I}_i$  is called the index space of variable  $x_i$ . Furthermore, in order to account for irregularities in programs, we allow quantified equations  $S_i [I]$  to have iteration dependent conditionals denoted by if conditional.

**Definition 3.3** (Iteration Dependent Conditional). A conditional  $\mathcal{C}^1(I)$  is called iteration dependent conditional of an equation and can be equivalently expressed by  $I \in \mathcal{I}_C \subseteq \mathbb{Z}^n$ , where the space  $\mathcal{I}_C$  is an iteration space called condition space.

A PLA is called *piecewise regular algorithm* (PRA) if the matrices  $P_i$  and  $Q_j$  are the identity matrix. Piecewise regular algorithms have only *regular or uniform dependencies*. However 45% of all programs are characterized by *affine or non-uniform dependencies*. The domains  $\mathcal{I}_i$  are defined as follows:

**Definition 3.4** (*Linearly Bounded Lattice*). A linearly bounded lattice denotes an index space of the form

$$\mathcal{I} = \{I \in \mathbb{Z}^n \mid I = M\kappa + c \wedge A\kappa \geq b\}$$

where  $\kappa \in \mathbb{Z}^l$ ,  $M \in \mathbb{Z}^{n \times l}$ ,  $c \in \mathbb{Z}^n$ ,  $A \in \mathbb{Z}^{m \times l}$  and  $b \in \mathbb{Z}^m$ .  $\{\kappa \in \mathbb{Z}^l \mid A\kappa \geq b\}$  denotes the set of integral points within a convex polyhedron or in case of boundedness within a polytope in  $\mathbb{Z}^l$ . This set is affinely mapped onto iteration vectors  $I$  using an affine transformation ( $I = M\kappa + c$ ).

Throughout the paper, we assume that the matrix  $M$  is square and of full rank. Then, each vector  $\kappa$  is uniquely mapped to an index point  $I$ . Furthermore, we require that the index space is bounded. Informally one can say the equations form the kernel of the loop nests whose range and bounds can be represented as LBLs. In the following example we illustrate the PLA notation with help of a FIR filter example.

**Example 3.1** The FIR (Finite Impulse Response) filter is described by the simple difference equation  $y(i) = \sum_{j=0}^{N-1} a(j) \cdot u(i-j)$  with  $0 \leq i < T$ ,  $N$  denoting the number of filter taps,  $a(j)$  the filter coefficients,  $u(i)$  the filter input, and  $y(i)$  the filter result. The difference equation on parallelization and embedding (i.e.  $a(0, j) = a(j)$ ,  $u(i-j, 0) = u(i-j)$ ) in a common index space can be written as the following PLA

$$\begin{aligned} a[i, j] &= a[0, j]; & u[i, j] &= u[i-j, 0]; & x[i, j] &= a[i, j] \cdot u[i, j]; \\ y[i, j] &= y[i, j-1] + x[i, j] \text{ if } j > 0; \end{aligned} \quad (2)$$

with the iteration domain  $\mathcal{I} = \{(i, j) \mid 0 \leq i \leq T-1 \wedge 0 \leq j \leq N-1\}$ .  $\text{if } j > 0$  is the iterative dependent conditional for the corresponding statement. The transformation localization converts affine dependencies into uniform dependencies by propagation of data from one index point to neighboring index point (see Fig. 2(b)). This enables a regular communication structure in hardware and enhances data reuse. After localization of variable  $u$ , the FIR filter has the following PLA description.

$$\begin{aligned} a[i, j] &= a[0, j] \\ u[i, j] &= u[i-1, j-1] \text{ if } i > 0 \wedge j > 0; & u[i, j] &= u[i-j, 0] \text{ if } i = 0 \vee j = 0 \\ x[i, j] &= a[i, j] \cdot u[i, j]; & y[i, j] &= y[i, j-1] + x[i, j] \text{ if } j > 0; \end{aligned} \quad (3)$$

## 4. Partitioning

Partitioning is a well known transformation which covers the index space of computation using congruent hyperplanes, hyperquaders, or parallelepipeds called *tiles*. The transformation has been studied in detail for compilers and its use has led to program acceleration through better cache reuse on sequential processors (i.e., *loop tiling or blocking*) [16], implementation of algorithms on given parallel architectures from supercomputers to multi-DSPs and FPGAs (Field Programmable Gate Arrays) [11]. It is carried out in order to match a loop nest implementation to resource constraints in terms of available number of processing elements (PEs), local memory, and communication bandwidth. Well known partitioning techniques are multi-projection, LSGP (local sequential global parallel, often also referred as clustering or blocking) and LPGS (local parallel global sequential, also referred as tiling). Formally, partitioning divides the index space  $\mathcal{I}$  using congruent tiles such that it is decomposed into spaces  $\mathcal{I}_1$  and  $\mathcal{I}_2$ , i.e.,  $\mathcal{I} \mapsto \mathcal{I}_1 \oplus \mathcal{I}_2$ <sup>1</sup>.  $\mathcal{I}_1 \in \mathbb{Z}^n$  represents the points within the tile and  $\mathcal{I}_2 \in \mathbb{Z}^n$  accounts for regular repetition

<sup>1</sup> $\mathcal{I}_1 \oplus \mathcal{I}_2 = \{i = i_1 + P \cdot i_2 \mid i_1 \in \mathcal{I}_1 \wedge i_2 \in \mathcal{I}_2 \wedge P \in \mathbb{Z}^{n \times n}\}$

of the tiles, i.e., the origin of each tile. In case of parallelepiped shaped tiles, tiles are defined by tiling matrix,  $P$ . Hierarchical partitioning methods have been studied in [5]. These partitioning techniques use different hierarchies of tiling matrices to divide the index space. Co-partitioning<sup>2</sup> is such an example of a 2-level hierarchical partitioning [5], where the index space is first partitioned into LS (local sequential) tiles, this tiled index space is tiled once more using GS (global sequential) tiles as shown in Fig. 2. Formally, it is defined as splitting of an index space into spaces  $\mathcal{I}_1, \mathcal{I}_2$  and  $\mathcal{I}_3$ , i.e.,  $\mathcal{I} \mapsto \mathcal{I}_1 \oplus \mathcal{I}_2 \oplus \mathcal{I}_3$ <sup>3</sup> using two congruent tiles defined by tiling matrices,  $P_{LS}$  and  $P_{GS}$ .  $\mathcal{I}_1 \in \mathbb{Z}^n$  represents the points within the LS tiles and  $\mathcal{I}_2 \in \mathbb{Z}^n$  accounts for the regular repetition of the origin of LS tiles (i.e., tiles marked with dashed line in Fig. 2a).  $\mathcal{I}_3 \in \mathbb{Z}^n$  accounts for the regular repetition of the GS tiles (i.e., bigger tiles marked with solid line in Fig. 2a). Similarly an  $n$ -hierarchical partitioning method splits the index space  $\mathcal{I}$  into  $n + 1$  spaces. The different partitioning schemes such as LSGP, LPGS, and co-partitioning are defined by specific scheduling which are typically realized through appropriate affine transformations defining the allocation and scheduling (see subsection 5.4).

**Example 4.1** *Co-partitioning of the iteration space of localized FIR filter example and FIR filter with no localization of variable  $a$  is shown in Fig. 2a) and 2b) respectively. The loop matrices used for tiling are*

$$P_{GS} = \begin{pmatrix} 4 & 0 \\ 0 & 6 \end{pmatrix} \quad P_{LS} = \begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix}$$

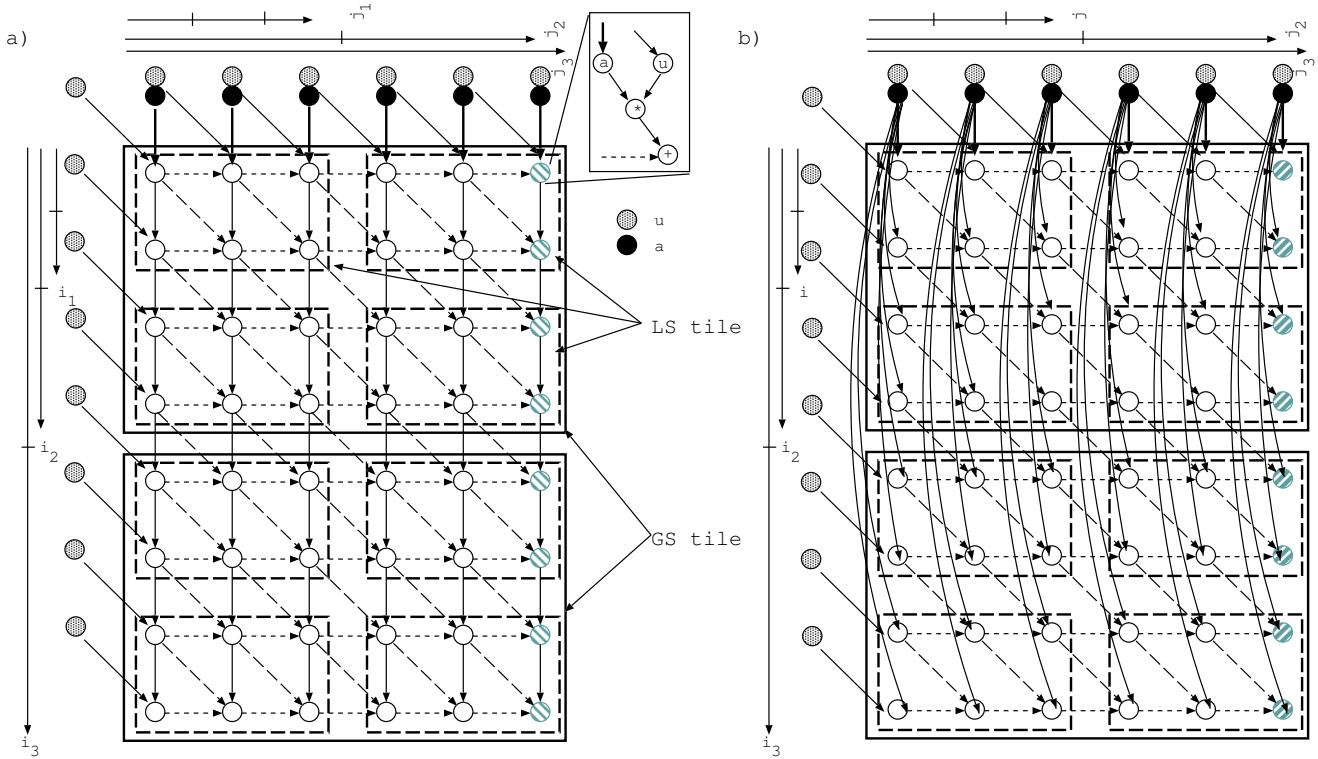
The new PLA obtained on application of co-partitioning on PLA given in Eq. 3 is given as following which can be also be attested by Fig. 2.

$$\begin{aligned}
a [i_1, j_1, i_2, j_2, i_3, j_3] &= \{ a[0, j_1, 0, j_2, 0, j_3] \\
&\quad \begin{cases} U_A(i_1 + 2 \cdot i_2 + 4 \cdot i_3 - (6 \cdot j_3)) & \text{if } j_1 = 0 \wedge j_2 = 0 \\ U_B(4 \cdot i_3 - (j_1 + 3 \cdot j_2 + 6 \cdot j_3)) & \text{if } i_1 = 0 \wedge j_1 > 0 \wedge i_2 = 0 \\ u[i_1 - 1, j_1 - 1, i_2, j_2, i_3, j_3] & \text{if } i_1 > 0 \wedge j_1 > 0 \\ u[i_1 + 1, j_1 - 1, i_2 - 1, j_2, i_3, j_3] & \text{if } i_1 = 0 \wedge j_1 > 0 \wedge i_2 > 0 \\ u[i_1 - 1, j_1 + 2, i_2, j_2 - 1, i_3, j_3] & \text{if } i_1 > 0 \wedge j_1 = 0 \wedge j_2 > 0 \\ u[i_1 + 1, j_1 + 2, i_2 - 1, j_2 - 1, i_3, j_3] & \text{if } i_1 = 0 \wedge j_1 = 0 \wedge i_2 > 0 \wedge j_2 > 0 \\ u[i_1 + 1, j_1 - 1, i_2 + 1, j_2, i_3 - 1, j_3] & \text{if } i_1 = 0 \wedge j_1 > 0 \wedge i_2 = 0 \wedge i_3 > 0 \\ u[i_1 + 1, j_1 + 2, i_2 + 1, j_2 - 1, i_3 - 1, j_3] & \text{if } i_1 = 0 \wedge j_1 = 0 \wedge i_2 = 0 \wedge j_2 > 0 \wedge i_3 > 0 \end{cases} \\
u [i_1, j_1, i_2, j_2, i_3, j_3] &= \\
x [i_1, j_1, i_2, j_2, i_3, j_3] &= a[i_1, j_1, i_2, j_2, i_3, j_3] \cdot u[i_1, j_1, i_2, j_2, i_3, j_3] \\
y [i_1, j_1, i_2, j_2, i_3, j_3] &= \begin{cases} 0 + x[i_1, j_1, i_2, j_2, i_3, j_3] & \text{if } j_1 = 0 \wedge j_2 = 0 \wedge j_3 = 0 \\ y[i_1, j_1 - 1, i_2, j_2, i_3, j_3] + x[i_1, j_1, i_2, j_2, i_3, j_3] & \text{if } j_1 > 0 \\ y[i_1, j_1 + 2, i_2, j_2 - 1, i_3, j_3] + x[i_1, j_1, i_2, j_2, i_3, j_3] & \text{if } j_1 = 0 \wedge j_2 > 0 \end{cases} \\
Y_{out} [i_1, j_1, i_2, j_2, i_3, j_3] &= y[i_1, j_1, i_2, j_2, i_3, j_3] \text{ if } j_3 = 0 \wedge j_2 = 1 \wedge j_1 = 2
\end{aligned} \tag{4}$$

for all  $I_1 = (i_1 \ j_1)^T \in \mathcal{I}_1, I_2 = (i_2 \ j_2)^T \in \mathcal{I}_2$ , and  $I_3 = (i_3 \ j_3)^T \in \mathcal{I}_3$ , with  $\mathcal{I}_1 = \{I_1 \in \mathbb{Z}^2 \mid 0 \leq i_1 < 2, 0 \leq j_1 < 3\}, \mathcal{I}_2 = \{I_2 \in \mathbb{Z}^2 \mid 0 \leq i_2 < 2, 0 \leq j_2 < 2\}$ , and  $\mathcal{I}_3 = \{I_3 \in \mathbb{Z}^2 \mid 0 \leq i_3 < 2, 0 \leq j_3 < 1\}$  One can verify that the index point  $I = (4, 3)$  is uniquely mapped to  $I_1 = (0, 0), I_2 = (0, 1)$  and  $I_3 = (1, 0)$  after co-partitioning. The dependency of variable,  $u$  leads to several new equations to account for embedding of equation in new index space defined by  $I_1, I_2$ , and  $I_3$ . e.g  $I = (4, 3)$  receives  $u$  value from  $I = (3, 2)$ . In the new index space of six dimensions,  $I_1 = (0, 0), I_2 = (0, 1)$  and  $I_3 = (1, 0)$  receives  $u$  value from  $I_1 = (1, 2), I_2 = (1, 0)$  and  $I_3 = (0, 0)$ . This is accounted by the last equation of  $u$ . The variable  $Y_{out}$  contains the final output value and  $U_A$  and  $U_B$  are the input values of variable  $u$  from memory.

<sup>2</sup>Co-partitioning uses both LSGP and LPGS methods in order to balance local memory requirements with I/O bandwidth with the advantage of problem size independence.

<sup>3</sup> $\mathcal{I}_1 \oplus \mathcal{I}_2 \oplus \mathcal{I}_3 = \{i = i_1 + P_{LS} \cdot i_2 + P_{GS} \cdot i_3 \mid i_1 \in \mathcal{I}_1 \wedge i_2 \in \mathcal{I}_2 \wedge i_3 \in \mathcal{I}_3 \wedge P_{LS}, P_{GS} \in \mathbb{Z}^{n \times n}\}$



**Figure 2. (a) Dependence graph of a localized co-partitioned FIR filter. (b) Dependence graph of a co-partitioned FIR filter where variable  $a$  is not localized. Furthermore, the dash filled circles are the iteration points producing the final output  $Y_{out}$ .**

Therefore, the problem to be dealt in the paper is that given an input PLA both with uniform and affine dependencies. How does one obtain an output PLA preserving the dependencies on hierarchical partitioning? And how to schedule the output PLA to obtain processor-time description for hardware generation. However the approach presented in the paper is not only limited to processor arrays but can be used for program analysis for parallelization. In the next section, we introduce our methodology for hierarchical partitioning of PLA.

## 5 Hierarchical Partitioning

The following methodology for partitioning proposed in this section encompasses all possible partitioning techniques (i.e. LSGP, LPGS, Co-partitioning, ...). The only major assumption is the standard use of congruent parallelepiped tiles for partitioning. Fig. 2(b) shows a partitioned index space with affine and uniform data dependencies. The first step *tiling* of index space which is equivalent to problem of *strip mining or loop tiling* in compiler theory. The only difference being is that resulting index space is a LBL. However, in our methodology we go one step further by embedding the data dependencies in new tiled index space. The advantage of this data dependence analysis step is that we remain in the polytope framework which offers possibility of mapping the algorithms onto massively parallel architectures. The modern compilers use the tiling transformation only to enhance data reuse and henceforth faster programs by reducing cache misses on uni-processor machine therefore no embedding of dependencies is done in modern compilers. The partitioning not only embeds the data dependencies

but also the iterative conditionals in the new index space. Furthermore the new data dependencies are also associated with unique iterative conditionals. Finally scheduling is an important step for describing the place and time co-ordinate of execution of each iteration in tiled index space and hardware generation. Therefore our approach for partitioning is constituted of the four steps *tiling*, *embedding* of data dependencies and control conditionals and finally *scheduling*. In following subsections, we discuss these ideas.

## 5.1 Tiling: Decomposition of Index Space

Similar to the idea that square tiling of a loop nest of depth 2 gives a loop nest of depth 4.  $n$ -Hierarchical partitioning tiles converts global iteration space of dimension  $m$  into  $(n + 1) \cdot m$  dimension index space ( for loop tiling  $n=1$ ). I.e. the global iteration space  $\mathcal{I}$  is decomposed into the direct sum of  $(n + 1)$  subspaces  $\mathcal{I}_1, \mathcal{I}_2, \dots$ , and  $\mathcal{I}_{n+1}$  such that  $\mathcal{I} \subseteq \mathcal{I}_1 \oplus \mathcal{I}_2 \oplus \dots \oplus \mathcal{I}_{n+1}$ .  $\mathcal{I}_1$  accounts for the index points in innermost tiles.  $\mathcal{I}_2$  accounts for the regular repetition of innermost tiles (i.e.  $\mathcal{I}_1$ ) and so on they collectively form the new index space. The tiles are parallelepiped and are described by the  $n$  tiling matrices  $(P_1, P_2, \dots, P_n)$  and a tiling offset  $q$  which describes the origin of tiling. If we assume the initial index space  $\mathcal{I}$  to be of the form  $\mathcal{I} = \{I | A \cdot I \geq b\}$ . Then the iteration bounds for new index space,  $\mathcal{I}_{new}$  can be represented in form of LBLs as following.

$$\mathcal{I}_{new} = \left\{ \begin{pmatrix} I_1 \\ I_2 \\ \vdots \\ I_{n+1} \end{pmatrix} = \begin{pmatrix} E & 0 & \dots & 0 \\ 0 & P_1 & \dots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \dots & P'_n \end{pmatrix} \begin{pmatrix} l_1 \\ l_2 \\ \vdots \\ l_{n+1} \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ \vdots \\ b \end{pmatrix} \wedge \begin{pmatrix} A_1 & 0 & \dots & 0 \\ 0 & A_2 & \dots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \dots & A_{n+1} \end{pmatrix} \begin{pmatrix} l_1 \\ l_2 \\ \vdots \\ l_{n+1} \end{pmatrix} \geq \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n+1} \end{pmatrix} \right\} \quad (5)$$

where,

$$A_1 = \begin{pmatrix} \sigma \cdot \text{adj}(P_1) \\ -\sigma \cdot \text{adj}(P_1) \end{pmatrix} \quad b_1 = \begin{pmatrix} 0 \\ (-\sigma \cdot \det(P_1) + 1) \cdot e \end{pmatrix} \quad A_2 = \begin{pmatrix} \sigma \cdot \text{adj}(P'_2) \\ -\sigma \cdot \text{adj}(P'_2) \end{pmatrix} \quad b_2 = \begin{pmatrix} 0 \\ (-\sigma \cdot \det(P'_2) + 1) \cdot e \end{pmatrix}$$

$\vdots$

$$A_n = \begin{pmatrix} \sigma \cdot \text{adj}(P'_n) \\ -\sigma \cdot \text{adj}(P'_n) \end{pmatrix} \quad b_n = \begin{pmatrix} 0 \\ (-\sigma \cdot \det(P'_n) + 1) \cdot e \end{pmatrix}$$

$$A_{n+1} I_{n+1} \geq b_{n+1} \equiv \text{Proj} \begin{pmatrix} A_n P'_n & A_n \\ 0 & A \end{pmatrix} \begin{pmatrix} I_n \\ I \end{pmatrix} \geq \begin{pmatrix} b_n + A_n \cdot q \\ b \end{pmatrix}. \text{ Also } \sigma = \frac{|\det(P_i)|}{\det(P_i)} \text{ and } P'_n = \prod_{i=1}^n P_i.$$

*Proj* defines the orthogonal projection over subspace defined by variable in  $I_n$  to eliminate all variables in  $I$ . It may be noted that above LBLs can also be written in terms of for loops.

**Example 5.1** *The iteration space of FIR filter example in section 3 ( $\mathcal{I} = \{(i, j)^T : 0 \leq i \leq 11, 0 \leq j \leq 5\}$ ) is co-partitioned (i.e. 2-hierarchical partitioning) with help of partitioning matrices  $P_1 = \begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix}$ ,  $P_2 = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$  and  $q = (0, 0)^T$  for inner, outer tiles, and offset respectively. This tiling in two dimension with only one partitioning matrix is popularly known as square tiling in compiler literature. Then on applying above formulas one obtains the new index space for FIR filter as shown in Eq. 6. On simplifying the bounds and removing redundant variables one can write the tiled iteration space in terms of for loop (assuming a sequential scheduling order) as following.*

*For( $i_1=0; i_1 \leq 1; i_1++$ )*

*For( $j_1=0; j_1 \leq 2; j_1++$ )*

*For( $i_2=0; i_2 \leq 1; i_2++$ )*

*For( $j_2=0; j_2 \leq 1; j_2++$ )*

*For( $i_3=0; i_3 \leq 1; i_3++$ )*

*For( $j_3=0; j_3 \leq 0; j_3++$ )*

$$\mathcal{I}_{new} = \left\{ \begin{pmatrix} i_1 \\ j_1 \\ i_2 \\ j_2 \\ i_3 \\ j_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 6 \end{pmatrix} \begin{pmatrix} i_1 \\ j_1 \\ i_2 \\ j_2 \\ i_3 \\ j_3 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \wedge \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & -2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 24 \\ 0 & 0 & 0 & 0 & 0 & -6 \\ 0 & 0 & 0 & 0 & 24 & 0 \\ 0 & 0 & 0 & 0 & -4 & 0 \end{pmatrix} \begin{pmatrix} i_1 \\ j_1 \\ i_2 \\ j_2 \\ i_3 \\ j_3 \end{pmatrix} \geq \begin{pmatrix} 0 \\ -5 \\ 0 \\ -5 \\ 0 \\ -3 \\ 0 \\ -3 \\ -23 \\ -5 \\ -23 \\ -11 \end{pmatrix} \right\} \quad (6)$$

However our tiling differs with loop tiling in compiler literature [16] in case of non-perfect tiling. Where we introduce dummy operations for invalid iteration points instead of introducing complex floor and division operations in the loop bounds. The reason being that extra operations are compensated by reduced loop overhead which is important for massively parallel implementation.

## 5.2 Embedding: Splitting of Dependencies

The existing affine and regular dependencies needs to be embedded in the new tiled iteration space. This step introduces new equation with new dependencies and iterative conditionals in case dependencies cross different tiles. All equations (as in Def. 3.1) are first brought in the form  $x[I] = \mathcal{F}(\dots, y[QI - d], \dots) \forall I \in I_c$  by transformation called *output normal form*. For the sake of simplicity above equation is written in equivalent form as following.

$$x[I] = \mathcal{F}(\dots, z[I], \dots) \forall I \in \mathcal{I}_c; \quad z[I] = y[QI - d] \forall I \in \mathcal{I}_c;$$

The purpose of embedding is to embed the equations in the new index space as following.

$$\begin{aligned} x[I_1, I_2, \dots, I_{n+1}] &= \mathcal{F}(\dots, z[I_1, I_2, \dots, I_{n+1}], \dots) \\ z[I_1, I_2, \dots, I_{n+1}] &= y[QI_1 - d - R_0, QI_2 - (R_1 - R_0), \dots, QI_n - (R_{n-1} - R_n), QI_{n+1} + R_n] \end{aligned}$$

One can write  $QI - d = Q(I_1 + I_2 + \dots + I_{n+1}) - d = (QI_1 - d - R_0) + (QI_2 - R_1 + R_0) + \dots + QI_n - (R_{n-1} - R_{n-2}) + QI_{n+1} + R_{n-1}$  (as all the  $R_i$  terms cancel each other), therefore the problem is to find all distinct  $(R_0, R_1, \dots, R_n)$ , s.t.  $QI_1 - d - R_0 \in \mathcal{I}_1, QI_2 - (R_1 - R_0) \in \mathcal{I}_2, \dots, QI_{n+1} + R_n \in \mathcal{I}_{n+1}$ . In [15], it was shown for  $n = 1$  (i.e. simple partitioning) one can setup a constraint polytope and enumerate all its point to find different possible values of  $R_0$ . In this section we introduce the method for embedding of dependencies in case of  $n$ -hierarchical partitioning. We first explain the extension with help of co-partitioning (i.e. 2-hierarchical partitioning).

Co-partitioning is partitioning done twice on an index space. Therefore there can be two approaches. In the first approach the PLA is partitioned twice. I.e.  $\mathcal{I} \Rightarrow \text{Tiling}(I, P_1) \Rightarrow \text{Embedding}(I_1, I_2) \Rightarrow \text{Tiling}(I_1, I_2, P_2) \Rightarrow \mathcal{I}_1 \oplus \mathcal{I}_2 \oplus \mathcal{I}_3 \Rightarrow \text{Embedding}(I_1, I_2, I_3)$ , therefore tiling and expand operations are applied twice. As enumeration is done twice the execution time of the approach is quite high and introduces redundant equations. However in the second approach the hierarchical partitioning is done directly. I.e.  $\mathcal{I} \mapsto \mathcal{I}_1 \oplus \mathcal{I}_2 \oplus \mathcal{I}_3$ , tiling and embedding operations are applied only once. Therefore on following second approach, we need to define embedding operation

for hierarchical partitioning. The embedding transformation for co-partitioning gives equation of the form.

$$\begin{aligned} x [I_1, I_2, I_3] &= \mathcal{F}(\dots, z [I_1, I_2, I_3], \dots) \\ z [I_1, I_2, I_3] &= y [QI_1 - d - R_0, QI_2 + R_0 - R_1, QI_3 + R_1] \end{aligned}$$

therefore the problem is to find all  $(R_0, R_1)$ , s.t. (a)  $QI_1 - d - R_0 \in \mathcal{I}_1$ , (b)  $QI_2 + R_0 - R_1 \in \mathcal{I}_2$ , and (c)  $QI_3 + R_1 \in \mathcal{I}_3$ . From Eq. 5 and (c) we infer  $I_3 = P'_2 \cdot l_3^1 + b \wedge A_3 \cdot l_3^1 \geq b_3$  and  $Q \cdot I_3 + R_1 = P'_2 \cdot l_3^2 + q$  (where  $\wedge A_3 \cdot l_3^2 \geq b_3$ ). Hence  $R_1$  must satisfy

$$R_1 = P_2 l_3^2 + q - Q(P_2 l_3^1 + q)$$

Similarly from Eq. 5 and (b), we infer  $I_2 = P'_1 \cdot l_2^1 \wedge A_2 \cdot l_2^1 \geq b_2$  and  $Q \cdot I_2 - R_1 + R_0 = P'_1 \cdot l_2^2$  (where  $A_2 \cdot l_2^2 \geq b_2$ ). Hence  $R_0$  must satisfy

$$R_0 = R_1 + P l_2^2 - Q(P l_2^1) = P l_2^2 - Q(P l_2^1) + P_2 l_3^2 + q - Q(P_2 l_3^1 + q)$$

Lastly,  $A_1 \cdot I_1 \geq b_1$  and  $A_1(Q \cdot I_1 - d - R_0) \geq b_1$  (from (a)). On replacing  $R_0$  we get following set of inequalities or *constraint polytope*

$$\begin{pmatrix} A_1 \cdot Q & A_1 Q \cdot P'_1 & -A_1 \cdot P'_1 & A_1 Q \cdot P'_2 & -A_1 \cdot P'_2 \\ A_1 & 0 & 0 & 0 & 0 \\ 0 & A_2 & 0 & 0 & 0 \\ 0 & 0 & A_2 & 0 & 0 \\ 0 & 0 & 0 & A_3 & 0 \\ 0 & 0 & 0 & 0 & A_3 \end{pmatrix} \begin{pmatrix} I_1 \\ l_2^1 \\ l_2^2 \\ l_3^1 \\ l_3^2 \end{pmatrix} \geq \begin{pmatrix} b_1 + A_1 \cdot q - A_1 Q \cdot q + A_1 d \\ b_1 \\ b_2 \\ b_2 \\ b_3 \\ b_3 \end{pmatrix}$$

The above polytope has  $5n$  variables and one must enumerate all its integral points which leads to distinct  $(R_0, R_1)$ . For each distinct value a new equation is generated. The enumeration is done by scanning the polytope for integer points lying in the rectangular hull of the polytope. The above argument can be extended by induction for  $n$ -hierarchical partitioning and gives following set of equations.

$$\begin{aligned} A_1 \cdot QI_1 + \sum_{i_1}^n A_1 Q \cdot P'_n l_n^1 - A_1 \cdot P'_n l_n^2 &\geq b_1 + A_1 \cdot c - A_1 Q \cdot c + A_1 d \\ A_1 I_1 &\geq b_1 \\ \forall 2 \leq i \leq n+1; A_i l_i^1 &\geq b_i \\ \forall 2 \leq i \leq n+1; A_i l_i^2 &\geq b_i \end{aligned}$$

Similarly by enumerating for each dependency ( $Q$ , and  $d$ ) and finding distinct  $(R_0, \dots, R_{n-1})$  one can add new equations to the partitioned description of the algorithm.

**Example 5.2** For our running FIR filter example one obtains the set of new equations as in table 1. Variable  $a$  with affine dependency yields only a single distinct  $(R_0, R_1)$ . whereas the variable  $u$  with uniform dependency  $((1 \ 1)^T)$  gives 6 distinct values of  $(R_0, R_1)$ . Therefore, the number of equations in obtained output PLA for variables  $a$ ,  $u$ , and  $y$  are 1, 6, and 3 respectively.  $E$  is the identity matrix. The input variables  $U_A$ , and  $U_B$  are also embedded in partitioned index space. In case of partitioning, the intuitive explanation of larger number of equations is due to dependencies crossing the tiles. The circuit interpretation of the program is that one need multiplexers to select the correct input. The control signals to the multiplexers are determined by the iterative control conditionals. The methodology to derive the corresponding control signals is discussed in the next subsection.

Equations	$Q$	$d$	$R_0$	$R_1$
$a[i_1, j_1, i_2, j_2, i_3, j_3] = a[0, j_1, 0, j_2, 0, j_3]$	$\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$	$(0\ 0)^T$	$(0\ 0)^T$	$(0\ 0)^T$
$u[i_1, j_1, i_2, j_2, i_3, j_3] = u[i_1 - 1, j_1 - 1, i_2, j_2, i_3, j_3]$	$E$	$(1\ 1)^T$	$(0\ 0)^T$	$(0\ 0)^T$
$u[i_1, j_1, i_2, j_2, i_3, j_3] = u[i_1 + 1, j_1 - 1, i_2 - 1, j_2, i_3, j_3]$	$E$	$(1\ 1)^T$	$(1\ 0)^T$	$(0\ 0)^T$
$u[i_1, j_1, i_2, j_2, i_3, j_3] = u[i_1 - 1, j_1 + 2, i_2, j_2 - 1, i_3, j_3]$	$E$	$(1\ 1)^T$	$(0\ 1)^T$	$(0\ 0)^T$
$u[i_1, j_1, i_2, j_2, i_3, j_3] = u[i_1 + 1, j_1 + 2, i_2 - 1, j_2 - 1, i_3, j_3]$	$E$	$(1\ 1)^T$	$(1\ 1)^T$	$(0\ 0)^T$
$u[i_1, j_1, i_2, j_2, i_3, j_3] = u[i_1 + 1, j_1 - 1, i_2 + 1, j_2, i_3 - 1, j_3]$	$E$	$(1\ 1)^T$	$(0\ 0)^T$	$(-1\ 0)^T$
$u[i_1, j_1, i_2, j_2, i_3, j_3] = u[i_1 + 1, j_1 + 2, i_2 + 1, j_2 - 1, i_3 - 1, j_3]$	$E$	$(1\ 1)^T$	$(0\ 1)^T$	$(-1\ 0)^T$
$x[i_1, j_1, i_2, j_2, i_3, j_3] = a[i_1, j_1, i_2, j_2, i_3, j_3] \cdot u[i_1, j_1, i_2, j_2, i_3, j_3]$	$E$	$(0\ 0)^T$	$(0\ 0)^T$	$(0\ 0)^T$
$y[i_1, j_1, i_2, j_2, i_3, j_3] = 0 + x[i_1, j_1, i_2, j_2, i_3, j_3]$	$E$	$(0\ 0)^T$	$(0\ 0)^T$	$(0\ 0)^T$
$y[i_1, j_1, i_2, j_2, i_3, j_3] = y[i_1, j_1 - 1, i_2, j_2, i_3, j_3] + x[\dots]$	$E$	$(0\ 1)^T$	$(0\ 0)^T$	$(0\ 0)^T$
$y[i_1, j_1, i_2, j_2, i_3, j_3] = y[i_1, j_1 + 2, i_2, j_2 - 1, i_3, j_3] + x[\dots]$	$E$	$(0\ 1)^T$	$(0\ 1)^T$	$(0\ 0)^T$
$Y_{out}[i_1, j_1, i_2, j_2, i_3, j_3] = y[i_1, j_1, i_2, j_2, i_3, j_3]$	$E$	$(0\ 1)^T$	$(0\ 0)^T$	$(0\ 0)^T$

**Table 1.** The table describes the set of new equations obtained on co-partitioning of the FIR filter.

### 5.3 Iteration dependent Conditionals

In this subsection we will transform the initial iteration conditionals in the tiled space. Furthermore embedding leads to new equations which in turn is associated with unique iterative conditionals. For  $n$ -Hierarchical partitioning each new equation has following new conditionals depending on corresponding  $(R_0, \dots, R_{n-1})$  as following because the conditions  $QI_1 - d - R_0 \in \mathcal{I}_1, QI_2 - (R_1 - R_0) \in \mathcal{I}_2, \dots, QI_{n+1} + R_n \in \mathcal{I}_{n+1}$  needs to be guaranteed. Therefore,

$$A_1(Q \cdot I_1 - d - R_0) \geq b_1, A_2(QI_2 + R_0 - R_1) \geq b_2, \dots, A_{n+1}(QI_{n+1} + R_n) \geq b_{n+1}. \quad (7)$$

The above conditionals contain lot of inequalities which is ugly for hardware implementation. As larger number of iterative conditionals means that control costs burden the computation for partitioned examples. However on removal of redundant inequalities (e.g. inequalities already defined by the iteration space) one obtains a simplified form for the iterative conditionals for practical examples. Furthermore assuming initial iterative conditionals ( $I \in \mathcal{I}_c$ ) is a LBL as following.

$$\mathcal{I}_c = \{I = A \cdot I + b \wedge A_s \cdot I \geq b_s\}$$

then the transformed conditional is as following

$$A_s A^{-1} \cdot I_1 + A_s A^{-1} \cdot I_2 + \dots + A_s A^{-1} \cdot I_{n+1} \geq b_s + A_s A^{-1} b$$

**Example 5.3** The table 2 summarizes the iterative control conditional obtained for the given FIR filter example. The conditionals correspond to the equations as given in table 1. These conditionals need to be evaluated by the processor array to control the multiplexers (see Fig. 3). Furthermore a counter is needed to provide the value of iteration variables. The methodology for efficient synthesis of control units has been proposed in [3]. The obtained control conditions are optimal because of removal of redundant equations. Once the control conditionals are obtained one obtains the output PLA as shown in example 4.1.

The set of equations in the example denotes the operations to be performed for a single iteration. These operations and iterations need to be scheduled on the processor array. This is done using an affine transformation realizing different partitioning schemes as discussed in the next section.

Variables	Control Conditionals	$Q$	$d$	$R_0$	$R_1$
$a$	Empty	$\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$	$(0\ 0)^T$	$(0\ 0)^T$	$(0\ 0)^T$
$u$	$i_1 > 0 \wedge j_1 > 0$	$E$	$(1\ 1)^T$	$(0\ 0)^T$	$(0\ 0)^T$
$u$	$i_1 = 0 \wedge j_1 > 0 \wedge i_2 > 0$	$E$	$(1\ 1)^T$	$(1\ 0)^T$	$(0\ 0)^T$
$u$	$i_1 > 0 \wedge j_1 = 0 \wedge j_2 > 0$	$E$	$(1\ 1)^T$	$(0\ 1)^T$	$(0\ 0)^T$
$u$	if $i_1 = 0 \wedge j_1 = 0 \wedge i_2 > 0 \wedge j_2 > 0$	$E$	$(1\ 1)^T$	$(1\ 1)^T$	$(0\ 0)^T$
$u$	if $i_1 = 0 \wedge j_1 > 0 \wedge i_2 = 0 \wedge i_3 > 0$	$E$	$(1\ 1)^T$	$(0\ 0)^T$	$(-1\ 0)^T$
$u$	if $i_1 = 0 \wedge j_1 = 0 \wedge i_2 = 0 \wedge j_2 > 0 \wedge i_3 > 0$	$E$	$(1\ 1)^T$	$(0\ 1)^T$	$(-1\ 0)^T$
$x$		$E$	$(0\ 0)^T$	$(0\ 0)^T$	$(0\ 0)^T$
$y$	if $j_1 = 0 \wedge j_2 = 0 \wedge j_3 = 0$	$E$	$(0\ 0)^T$	$(0\ 0)^T$	$(0\ 0)^T$
$y$	if $j_1 > 0$	$E$	$(0\ 1)^T$	$(0\ 0)^T$	$(0\ 0)^T$
$y$	if $j_1 = 0 \wedge j_2 > 0$	$E$	$(0\ 1)^T$	$(0\ 1)^T$	$(0\ 0)^T$
$Y_{out}$	if $j_3 = 0 \wedge j_2 = 1 \wedge j_2 = 2$	$E$	$(0\ 1)^T$	$(0\ 0)^T$	$(0\ 0)^T$

**Table 2. The iterative control conditional for the co-partitioned FIR filter example.**

## 5.4 Scheduling

Linear transformations are used as *space-time mappings* in order to assign a processor  $p$  (space) and a sequencing index  $t$  (time) to index vectors [6]. In LSGP, all index points within a tile are executed sequentially by the same processor and the index points in different tiles are executed in parallel by different processors. Therefore, the number of processors is equal to the number of tiles. In LPGS, all index points within a tile are executed in parallel whereas the tiles are executed sequentially. This observation is incorporated into the affine transformation defining the space-time mapping for both LSGP and LPGS methods as following.

**Definition 5.1** (*Space-time mapping for LSGP and LPGS*)

$$\begin{pmatrix} p \\ t \end{pmatrix} = \begin{pmatrix} 0 & E \\ \lambda_1 & \lambda_2 \end{pmatrix} \begin{pmatrix} I_1 \\ I_2 \end{pmatrix} \text{ (LSGP)} \quad \begin{pmatrix} p \\ t \end{pmatrix} = \begin{pmatrix} E & 0 \\ \lambda_1 & \lambda_2 \end{pmatrix} \begin{pmatrix} I_1 \\ I_2 \end{pmatrix} \text{ (LPGS)}$$

where  $E$  is the identity matrix,  $\lambda_1 \in \mathbb{Z}^{1 \times n_1}$ ,  $\lambda_2 \in \mathbb{Z}^{1 \times n_2}$  and  $n_1 + n_2 = n$ .  $p$  defines the processor index and  $t$  defines the time step of execution.

Similarly in co-partitioning, the index points within the LS tiles are executed sequentially. All the LS tiles within a GS tile are executed in parallel by the processor array. Therefore, the number of processors in the array is equal to the number of LS tiles within a GS tile. The GS tiles are executed sequentially.

**Definition 5.2** (*Space-time mapping for co-partitioning*). A space-time mapping in case of co-partitioning is an affine transformation of the form

$$\begin{pmatrix} p \\ t \end{pmatrix} = \begin{pmatrix} 0 & E & 0 \\ \lambda_1 & \lambda_2 & \lambda_3 \end{pmatrix} \begin{pmatrix} I_1 \\ I_2 \\ I_3 \end{pmatrix} \quad (8)$$

where  $E \in \mathbb{Z}^{n_2 \times n_2}$  is the identity matrix,  $\lambda_1 \in \mathbb{Z}^{1 \times n_1}$ ,  $\lambda_2 \in \mathbb{Z}^{1 \times n_2}$ ,  $\lambda_3 \in \mathbb{Z}^{1 \times n_3}$ .

Similarly, other hierarchical partitioning schemes can be realized using appropriate selection of affine transformation characterizing the scheduling and the allocation of the index points. The problem of determining an optimal sequencing index (i.e.,  $\lambda_1, \lambda_2, \dots$ ) taking into account constraints on timing of PAs and availability of resources is solved by a Mixed Integer Linear Programming (MILP) formulation of the problem in [7].

**Example 5.4** For FIR filter a feasible space-time mapping of the co-partitioned PLA is as shown in Eq. 9. The execution of the iteration points corresponding to the given schedule as shown in Fig. 3(a). The schedule defines the co-partitioning scheme, where iteration within the smaller tile are executed sequentially on the same processor. Whereas the smaller tiles are executed in parallel on different processors.

$$\begin{pmatrix} p \\ t \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 3 & 1 & 4 & 3 & 8 & 0 \end{pmatrix} \begin{pmatrix} i_1 \\ j_1 \\ i_2 \\ j_2 \\ i_3 \\ j_3 \end{pmatrix} \quad (9)$$

The description obtained on applying the above affine transformation to the output PLA in example 4.1 is as shown in Eq. 10. One can derive the description of the processor array architecture from the following PLA. The obtained processor array architecture is shown in Fig. 3(b).

$$\begin{aligned} a[p_1, p_2, t] &= \{ a[0, j_1, 0, j_2, 0, j_3] \\ u[p_1, p_2, t] &= \begin{cases} U_A(i_1 + 2 \cdot i_2 + 4 \cdot i_3 - (6 \cdot j_3)) & \text{if } j_1 = 0 \wedge j_2 = 0 \\ U_B(4 \cdot i_3 - (j_1 + 3 \cdot j_2 + 6 \cdot j_3)) & \text{if } i_1 = 0 \wedge j_1 > 0 \wedge i_2 = 0 \\ u[p_1, p_2, t - 4] & \text{if } i_1 > 0 \wedge j_1 > 0 \\ u[p_1 - 1, p_2, t - 2] & \text{if } i_1 = 0 \wedge j_1 > 0 \wedge p_1 > 0 \\ u[p_1, p_2 - 1, t - 4] & \text{if } i_1 > 0 \wedge j_1 = 0 \wedge p_2 > 0 \\ u[p_1 - 1, p_2 - 1, t - 2] & \text{if } i_1 = 0 \wedge j_1 = 0 \wedge p_1 > 0 \wedge p_2 > 0 \\ u[p_1 + 1, p_2, t - 2] & \text{if } i_1 = 0 \wedge j_1 > 0 \wedge p_1 = 0 \wedge i_3 > 0 \\ u[p_1 + 1, p_2 - 1, t - 2] & \text{if } i_1 = 0 \wedge j_1 = 0 \wedge p_1 = 0 \wedge p_2 > 0 \wedge i_3 > 0 \end{cases} \\ x[p_1, p_2, t] &= a[p_1, p_2, t] \cdot u[p_1, p_2, t] \\ y[p_1, p_2, t] &= \begin{cases} 0 + x[p_1, p_2, t] & \text{if } j_1 = 0 \wedge p_2 = 0 \wedge j_3 = 0 \\ y[p_1, p_2, t - 1] + x[p_1, p_2, t] & \text{if } j_1 > 0 \\ y[p_1, p_2 - 1, t - 1] + x[p_1, p_2, t] & \text{if } j_1 = 0 \wedge p_2 > 0 \end{cases} \\ Y_{out}[p_1, p_2, t] &= y[p_1, p_2, t] \text{ if } j_3 = 0 \wedge p_2 = 1 \wedge j_1 = 2 \end{aligned} \quad (10)$$

The resulting architecture is a  $2 \times 2$  processor array. The white boxes denote the registers. The number of registers can be obtained by multiplying the schedule vector with the dependency vector. E.g. a vector dependency of  $u$ ,  $(1 \ 1 \ 0 \ 0 \ 0 \ 0)^T$  leads to connection with four registers. The processing element  $PE(0, 0)$  executes the iterations in the smaller tile at  $(0, 0)$  within the bigger tiles.

Therefore, on summarizing the above subsections one can partition and schedule a given PLA with the following algorithm.

---

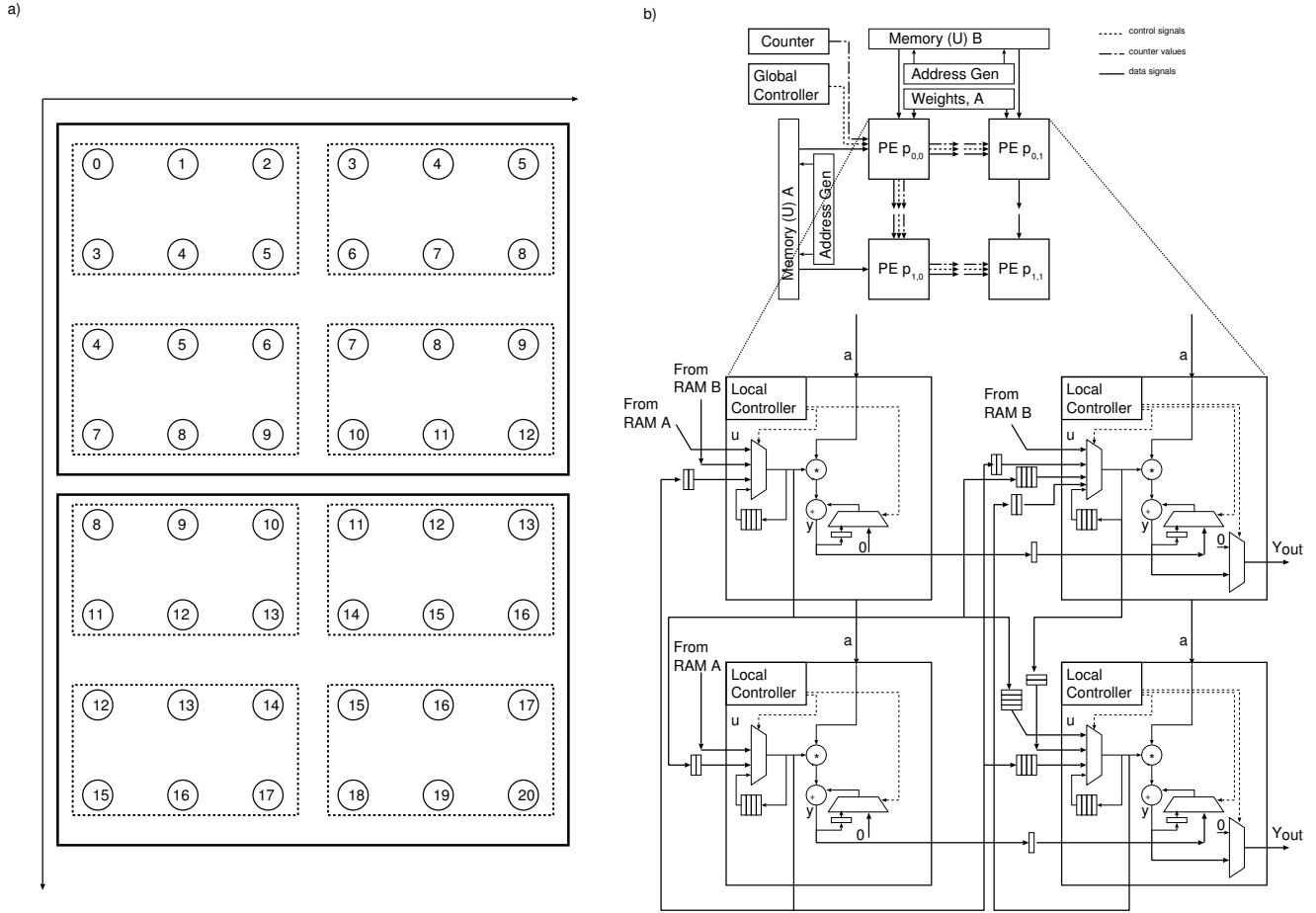
ALGORITHM:  $n$ -Hierarchical Partitioning

---

```

DO TILING( $\mathcal{I}, n, P_1, \dots, P_n$ )
  FORALL equations  $S_i$ 
    FORALL variables  $x_i$  with affine dependencies
      determine  $(R_0, \dots, R_{n-1})$ 
      FOR each distinct  $(R_0, \dots, R_{n-1})$ 
        Compute conditional space.
        Update the variables index, dependencies, and conditionals
      ENDFOR
  ENDFOR

```



**Figure 3. a) The iteration space of co-partitioned FIR filter example without the dependencies. The number denotes the start time of the iterations as defined the affine schedule in example 5.4 b) The corresponding processor array architecture for the FIR filter.**

```

ENDFOR
ENDFOR
OD

```

## 6 Conclusions and Future Directions

In this paper, we presented a exact methodology for partitioning of piecewise linear algorithms (i.e. perfectly nested loop programs) with consideration of both affine and uniform dependencies. This enables not only mapping of algorithms onto massively parallel architectures but is also of interest in studying the additional control introduced due to partitioning. The transformation has been implemented in *PARO* design system [12]. The future works entails studying techniques for efficient enumeration of the index space for reduced execution time of the program.

## References

- [1] F. Catthoor, K. Danckaert, S. Wuytack, and N. D. Dutt. Code Transformations for Data Transfer and Storage Exploration Preprocessing in Multimedia Processors. *IEEE Design and Tests of Computers*, 18(3):70–82, 2001.
- [2] S. Derrien and T. Risset. Interfacing Compiled FPGA Programs: The MMAAlpha Approach. In *PDPTA*, 2000.
- [3] H. Dutta, F. Hannig, and J. Teich. Controller Synthesis for Mapping Partitioned Programs on Array Architectures. In *Proceedings of the 19th International Conference on Architecture of Computing Systems (ARCS 2006), Frankfurt / Main, Germany*, pages 176–191, Frankfurt, Germany, Mar. 2006. Springer.
- [4] 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.
- [5] U. Eckhardt and R. Merker. Hierarchical Algorithm Partitioning at System Level for an Improved Utilization of Memory Structures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(1):14–24, 1999.
- [6] F. Hannig, H. Dutta, and J. Teich. Regular Mapping for Coarse-grained Reconfigurable Architectures. In *Proceedings of the 2004 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2004)*, volume V, pages 57–60, Montréal, Quebec, Canada, May 2004. IEEE Signal Processing Society.
- [7] F. Hannig and J. Teich. Design Space Exploration for Massively Parallel Processor Arrays. In V. Malyskhin, editor, *Parallel Computing Technologies, 6th International Conference, PaCT 2001, Proceedings*, volume 2127 of *Lecture Notes in Computer Science (LNCS)*, pages 51–65, Novosibirsk, Russia, Sept. 2001. Springer.
- [8] R. Keryell, C. Ancourt, F. Coelho, B. Creusillet, F. cois, and I. Pierre. Pips: a workbench for building interprocedural parallelizers, 1996.
- [9] R. Kuhn. Transforming Algorithms for Single-Stage and VLSI Architectures. In *Workshop on Interconnection Networks for Parallel and Distributed Processing*, pages 11–19, West Lafayette, IN, Apr. 1980.
- [10] 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.
- [11] J. Oldfield and R. Dorf. *Field Programmable Gate Arrays: Reconfigurable Logic for Rapid Prototyping and Implementation of Digital Systems*. John Wiley & Sons, Chichester, New York, 1995.
- [12] PARO Design System Project. [www12.informatik.uni-erlangen.de/research/paro](http://www12.informatik.uni-erlangen.de/research/paro).
- [13] Synfora, Inc. [www.synfora.com](http://www.synfora.com).
- [14] J. Teich. *A Compiler for Application-Specific Processor Arrays*. PhD thesis, Institut für Mikroelektronik, Universität des Saarlandes, Saarbrücken, Deutschland, September 1993.
- [15] J. Teich and L. Thiele. Exact Partitioning of Affine Dependence Algorithms. In E. Deprettere, J. Teich, and S. Vassiliadis, editors, *Embedded Processor Design Challenges*, volume 2268 of *Lecture Notes in Computer Science (LNCS)*, pages 135–153, Mar. 2002.
- [16] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Inc., 1996.