

Synthesis of FPGA Implementations from Loop Algorithms

M. Bednara

Computer Engineering Laboratory
Department of EE and IT
University of Paderborn
Paderborn, Germany

J. Teich*

Computer Engineering Laboratory
Department of EE and IT
University of Paderborn
Paderborn, Germany

Abstract *We consider the problem of automatically mapping computation-intensive loop nests onto FPGA hardware. The regular cell array structure of these chips reflects the parallelism in regular loop-like computations. Furthermore, the flexibility of FPGAs allows the cost-effective implementation of reconfigurable high performance processor arrays. So far, there exists no continuous design flow that allows automated generation of FPGA configuration data from a loop nest specified in a high level language. Here, we present a methodology for automatic generation of synthesizable VHDL code specifying a processor array and optimized for FPGA implementation.*

Keywords: FPGA, Processor Arrays, Hardware Mapping

1 Introduction

In the past, most of the systematic work in the area of systolic processor arrays was only at the pure algorithmic level, by proposing transformations, e.g., for space-time mappings, with almost no support for the synthesis of transformed loop nests in silicon. Therefore, most of the design time was spent in entering a processor design and simulating it, taking often many years to complete a systolic processor array. For these reasons, the interest in VLSI processor arrays decreased rapidly during the last decade.

FPGA-based processor arrays. With the FPGA technology in mind, targeting of loop nests to reconfigurable processor array hardware will be very

beneficial for the following reasons:

- Modern embedded system applications require large computation performance, low power dissipation and small area. Processor arrays fulfill these constraints.
- Processor arrays directly reflect the regular structure of the loop nest, while the FPGA directly reflects the regular structure of the processor array.
- The reconfigurability facilitates to rapidly change the application which is essential for embedded systems with resource constraints.

Contribution. The compile time for mapping a loop program to hardware still remains a critical factor even with pre-fabricated FPGA chips at hand.

In [1], we could impressively demonstrate the potential of FPGA-based regular array coprocessors. We used, however, a design that was only slightly optimized for FPGA technology. Here, we describe the backend of the design tool PARO (*P*iecewise *A*ffine *A*lgorithm and *A*rchitecture *R*esearch and *O*ptimization), which is summarized in Section 2. We propose solutions to the currently missing automation of the hardware synthesis process and show, how VHDL specifications may be generated and which assumptions must be made for an algorithm and given space-time mapping to be mappable to hardware.

Section 2 gives a very short summary of the the PARO design system. In Section 3, we describe the class of loop algorithms which we synthesize working FPGA-implementations from. The code generation itself is described in Section 4 (array structure) and Section 5 (processor elements).

*This work is supported by DFG Sonderforschungsbereich 376 "Massive Parallelität."

2 The PARO design system

The intention of PARO is to automatically map nested loop programs onto reconfigurable hardware.

Therefore, a loop nest must be parallelized first. Here, we use Feautrier's technique [2] for converting imperative loop programs to single assignment code (this front-end is similar to the approach in [3]). The parallelized algorithm is now subject to several equivalence-preserving transformations such as *localization of data dependences*, *affine index transformations*, *partitioning* [11, 12, 13], and *automatic control generation* [10].

For the generation of synthesizable HDL code (performed by the PARO back-end), we assume that the algorithm is already partitioned and localized.

3 Requirements for Hardware Mapping

In this section, we define a *piecewise regular algorithm* and develop necessary conditions for these algorithms in order to be hardware-mappable.

3.1 Locality

We assume that affine data dependences have been localized resulting in a *piecewise regular algorithm* that is defined as follows:

Definition 3.1 A piecewise regular algorithm \mathcal{A} contains N quantified equations $S_1[I], \dots, S_i[I], \dots, S_N[I]$. Each equation $S_i[I]$ is of the form

$$x_i[I] = f_i(\dots, x_j[I - d_{ji}], \dots)$$

where $I \in I_i \subseteq \mathbb{Z}^n$, $x_i[I]$ are indexed variables, f_i are arbitrary functions, $d_{ji} \in \mathbb{Z}^n$ are constant data dependence vectors, and \dots denote similar arguments.

The domains I_i are called index spaces, and in our case defined as follows:

Definition 3.2 (*Linearly Bounded Lattice*). A linearly bounded lattice denotes an index space of the form

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

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 $b \in \mathbb{Z}^m$. $\{\kappa \in \mathbb{Z}^l \mid A\kappa \geq b\}$ defines an integral convex polyhedron or a polytope in \mathbb{Z}^l . This set is affinely mapped onto iteration vectors I using an affine transformation ($I = M\kappa + c$).

Throughout this paper, we assume that the matrix M is square and invertible. Then, each vector κ is uniquely mapped to an index point I . In this case, we can rewrite the index space as $I = \{I \in \mathbb{Z}^n \mid A'I \geq b'\}$ where $A' = AM^{-1}$ and $b' = AM^{-1}c + b$. Furthermore, we require that the index space is bounded.

Example 3.1 Consider a piecewise regular algorithm which consists of three quantified indexed equations $S_1[I] \cdots S_3[I]$:

$$\begin{aligned} y[i, j, k] &= (y[i, j-1, k-1] \cdot u[i, j, k-2]) \\ &\quad + (a[i-1, j-1, k-1] \cdot u[i, j-1, k]) \\ a[i, j, k] &= y[i, j-1, k] - a[i-1, j, k] \\ u[i, j, k] &= u[i, j-1, k-1] \\ &\quad - (y[i-1, j-1, k] \cdot y[i-1, j, k-4]) \\ &\quad \forall (i \ j \ k)^T = I \in I \end{aligned}$$

The index space I is common to all 3 quantified equations ($I = I_1 = I_2 = I_3$) and given by

$$I = \left\{ I \in \mathbb{Z}^3 \mid \begin{pmatrix} -1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & -1 & 0 \\ 1 & 1 & 0 \\ 1 & -1 & 0 \\ -1 & -1 & 0 \\ 0 & 0 & -1 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \end{pmatrix} \leq \begin{pmatrix} -1 \\ 6 \\ -2 \\ 12 \\ 2 \\ -4 \\ 0 \\ 8 \end{pmatrix} \right\}.$$

Computations of piecewise regular algorithms may be represented by a *dependence graph* (DG) (Fig. 1) which expresses the partial order between the operations.

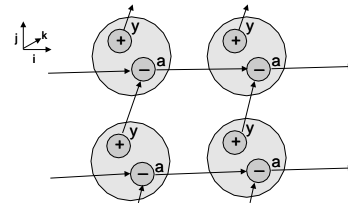


Figure 1: Segment of the DG of the algorithm in Example 3.1. Here, only dependences for variable a are depicted.

Each variable of the algorithm is represented at every index point $I \in I$ by one DG node, while the

edges correspond to the data dependences. The DG implicitly specifies all legal execution orderings of operations: on a directed path in the dependence graph from one node $a[J]$ to a node $c[K]$, the computation of $a[J]$ must precede the computation of $c[K]$.

3.2 Homogeneous index space

Without loss of generality we assume that all indexed variables are embedded in a common index space I ($I_1 = I_2 = \dots = I_N$).

3.3 Causality and Uniqueness

Linear transformations as in Eq. (1), are used as *space-time mappings* [5, 7] in order to assign a *processor index* $p \in \mathbb{Z}^{n-1}$ (space) and a *sequencing index* $t \in \mathbb{Z}$ (time) to index vectors $I \in I$.

$$\begin{pmatrix} p \\ t \end{pmatrix} = \begin{pmatrix} Q \\ \lambda \end{pmatrix} I = TI \quad (1)$$

In Eq. (1), $Q \in \mathbb{Z}^{(n-1) \times n}$ and $\lambda \in \mathbb{Z}^{1 \times n}$. The main reasons for using linear allocation and scheduling functions is that the data flow between PEs is local and regular which is essential for hardware implementations. The interpretation of such a linear transformation is as follows: The set of operations defined at index points $\lambda \cdot I = \text{const.}$ are scheduled at the same time step. The index space of allocated processing elements (*processor space*) is denoted by the set $Q = \{p \mid p = Q \cdot I \wedge I \in I\}$. This set can also be obtained by choosing a projection of the dependence graph along a vector $u \in \mathbb{Z}^n$, i.e. any coprime¹ vector u satisfying $Q \cdot u = 0$ describes the allocation equivalently [5]. Allocation and scheduling must satisfy that no data dependences in the DG are violated. This is ensured by the *causality constraint* $\lambda \cdot d_{ji} \geq 0$ for all data dependences d_{ji} . A sufficient condition for guaranteeing that no two or more index points are assigned to a processing element at the same time step is given by $\text{rank}(T) = n$. Using the projection vector u satisfying $Q \cdot u = 0$, this condition is equivalent to $\lambda \cdot u \neq 0$.

¹A vector x is said to be *coprime* if the absolute value of the greatest common divisor of its elements is one.

3.4 Iterative Schedules

Operations of subsequent index points mapped to the same PE are typically scheduled each *iteration interval*² P , $P = |\lambda \cdot u| \geq 1$ instead of each clock cycle. Thereby, operations of subsequent index points may overlap execution, see Fig. 2, for instance. This is called *functional pipelining*.

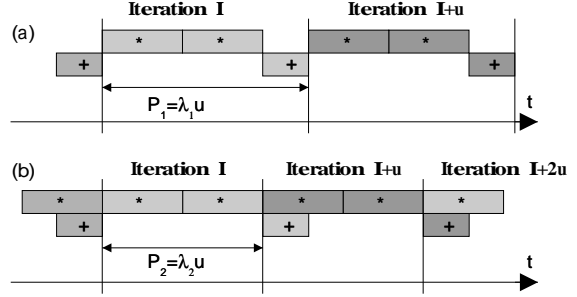


Figure 2: Cyclic schedule of Equation $S_1[I]$ of Example 3.1. Iteration period P_1 (a) is without and P_2 (b) with functional pipelining.

For finding a latency-minimal schedule for a given projection vector u under the constraint of limited resources, we solve a mixed integer linear program (MILP) as introduced in [14, 12]. The results of this MILP are used for the synthesis of the local PE controller (see Section 5.4).

3.5 Space-Time Independence

Scheduling and projection of algorithm \mathcal{A} results in an algorithm \mathcal{A}^* defined over an index space $I^* = \{I^* \in \mathbb{Z}^n \mid A^* I^* \geq b^*\}$ with $I^* = (p \ t)^T$, $A^* = A' \cdot T^{-1}$, and $b^* = b'$.

In a synchronous processor array, all PEs start at time step zero and execute until the last computation of the array is finished. Thus, the projected index space I^* must be a *prism* meaning that the processor index p must not depend on the time index and vice versa.

Hence, A^* must be decomposable into submatrices A_p and A_t , and b^* must be decomposable

²The iteration interval P is the number of time steps between the evaluation of two successive instances of a variable within one processing element [14].

into sub-vectors b_p and b_t such that

$$I^* = \left\{ \begin{pmatrix} p \\ t \end{pmatrix} \in \mathbb{Z}^n : \begin{pmatrix} A_p & 0 \\ 0 & A_t \end{pmatrix} \cdot \begin{pmatrix} p \\ t \end{pmatrix} \geq \begin{pmatrix} b_p \\ b_t \end{pmatrix} \right\} \quad (2)$$

Example 3.2 In Example 3.1, processor space and time are independent. We have $A_t = (-1 \ 1)^T$, $b_t = (0 \ 8)^T$, and

$$A_p = \begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 1 & 1 \\ 1 & -1 \\ -1 & -1 \end{pmatrix}, b_p = \begin{pmatrix} -1 \\ 6 \\ -2 \\ 12 \\ 2 \\ -4 \end{pmatrix}.$$

The prism form of the index space can always be achieved by a transformation (*extension of index spaces*) described in [9].

4 Array Synthesis

In this section we present our approach for generating VHDL code for the array structure, which is completely structural and contains only PE component instances and wiring of these instances.

4.1 PE Instantiation

The processor space Q as defined in Section 3.3 can be a convex polytope of any shape. The processor space Q of the algorithm introduced in Example 3.1 is shown in Fig. 3.

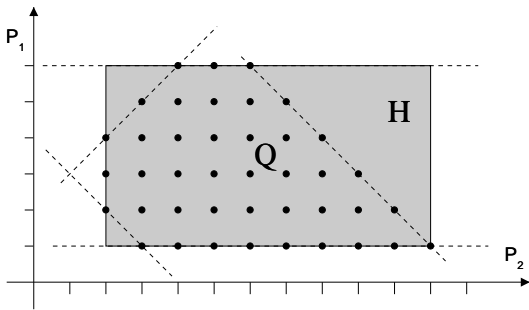


Figure 3: Processor space Q and its rectangular hull \mathcal{H}

For instantiating and connecting the PE components, we use a *GENERATE* loop construct. This allows interconnecting PE instances via signal arrays as described later in this section. In the code example in Fig. 4 the nested loops L1 and L2 span

```
L1: FOR p1 IN H1_min TO H1_max GENERATE
  L2: FOR p2 IN H2_min TO H2_max GENERATE
    I: IF (inside_I(p1, p2) = 1) GENERATE

      PE_Map: PE_component
      PORT MAP (...);

    END GENERATE I;
  END GENERATE L2;
END GENERATE L1;
```

Figure 4: *GENERATE* loop construct for PE component instantiation

a rectangular hull \mathcal{H} around the processor space Q , as depicted in Fig. 3. \mathcal{H} is defined as follows:

$$\mathcal{H} = \{(p_1, p_2) \in \mathbb{Z}^2 : l_1 \leq p_1 \leq u_1, l_2 \leq p_2 \leq u_2\} \quad (3)$$

where the lower bounds l_i , $i = 1, \dots, n-1$ are given by $l_i = \min\{p_i | p = (p_1 \dots p_i \dots p_{n-1})^T \in Q\}$ and the upper bounds u_i are defined using the max function accordingly. In general $\mathcal{H} \neq Q$, so the rectangular hull contains index points where no PE must be generated. To avoid PE instantiation here, we define an index check function *inside_I* that is evaluated in each iteration of the inner loop and returns 1 iff the index point represented by the current loop indices is a valid index point of Q . Otherwise, 0 is returned.

Example 4.1 Example 3.1 continued. The corresponding function *inside_I* of the processor space Q is shown in Fig. 5.

```
FUNCTION inside_I(p1, p2: INTEGER)
RETURN INTEGER IS
  VARIABLE valid: INTEGER;
BEGIN
  IF ((p1 >= 1) AND (p1 <= 6) AND
      (p2 => 2) AND (p1+p2 <= 12) AND
      (p1-p2 <= 2) AND (p1+p2 => 4) )
  THEN valid:=1; ELSE valid:=0;
  END IF;
  RETURN (valid);
END;
```

Figure 5: Index check function of processor space in Example 3.1.

4.2 Interconnect: Signal Arrays

Using the space time mapping T of Eq. 1, any dependence vector d of a quantified equation is trans-

formed into a vector d^* as follows:

$$d^* = (d_p^* \ d_t^*)^T = T \cdot d \quad (5)$$

where $d_p^* \in \mathbb{Z}^{n-1}$, and $d_t \in \mathbb{N}_0$. d_p^* denotes the *processor displacement* and d_t the *time displacement*.

The interconnect topology is thus given by the processor displacement vectors d_p^* . In our methodology, an $n - 1$ dimensional VHDL signal array is defined for each vector d_p^* and *GENERATE* loops are used to build the inter-PE interconnect structure. Each array element is a signal vector that connects two PEs.

Example 4.2 *Example 3.1 continued. Consider the transformed equation $S_2[I^*]$:*

$$a[p_1, p_2, t] = y[p_1, p_2 - 1, t] + a[p_1 - 2, p_2, t] \quad (6)$$

Variables a and y are 16 bit numbers here. We declare two-dimensional signal arrays for a and y (Fig. 6) defined in the range of \mathcal{H} . This array contains signals that are never used. Those, however, will be optimized away by logic synthesis tools. The PEs are interconnected by accessing the sig-

```

TYPE T0 IS STD_LOGIC_VECTOR (0 TO 15);
TYPE T1 IS ARRAY
  (NATURAL RANGE 1 TO 6) OF T0;
TYPE T2 IS ARRAY
  (NATURAL RANGE 2 TO 11) OF T1;
SUBTYPE array_T IS T2;

SIGNAL a, y: array_T;
...

PE_Map: PE_component
PORT MAP ( y_in => y(p1)(p2-1),
           a_in => a(p1-2)(p2),
           a_out => a(p1)(p2),
           ...
           );

```

Figure 6: Signal array declaration and signal association to a PE instance.

nal array via the loop indices p1 and p2, refer to Fig. 6.

4.3 Arrays Borders

At the array borders, the array signals are connected to the array ports in a similar way as to the PE instances. We declare no explicit border processors. It is the task of the environment to ensure

that all data is available at the right time step and processor index.

5 Synthesis of Processor Elements

The processor elements may be synthesized separately from the array structure.

5.1 Operator Splitting Transformation

In order to map computations in quantified equations to hardware resources, it is necessary to split computations at the right hand side of equations into simple expressions that may be handled by available resources. This step, *operator splitting*, must be performed prior to space-time mapping (scheduling).

Example 5.1 *We transform each of the equations $S_i[I]$ of Example 3.1 into a set of m_i new equations $S_{i,0}[I] \cdots S_{i,m_i-1}[I]$ such that each new equation consist of only one operation on its right side:*

$$\begin{aligned}
S_{0,0}[I] : \quad & y[i, j, k] = y_1[i, j, k] + y_2[i, j, k] \\
S_{0,1}[I] : \quad & y_1[i, j, k] = y[i, j - 1, k - 2] \cdot u[i, j, k - 2] \\
S_{0,2}[I] : \quad & y_2[i, j, k] = a[i - 1, j - 1, k - 1] \cdot u[i, j - 2, k] \\
S_{1,0}[I] : \quad & a[i, j, k] = y[i, j - 1, k] - a[i - 1, j, k] \\
S_{2,0}[I] : \quad & u[i, j, k] = u[i, j - 1, k - 1] - u_1[i, j, k] \\
S_{2,1}[I] : \quad & u_1[i, j, k] = y[i - 1, j - 1, k] \cdot y[i - 1, j, k - 4]
\end{aligned}$$

y_1 , y_2 , and u_1 are three new indexed variables.

Let \mathcal{S} denote the set of all equations $S_{i,j}[I]$ defined at index point I . Each right hand side expression now represents a unit that may be scheduled and bound to a hardware resource. In the following, we use the terms *variable* and *operator* synonymously.

5.2 Resource Allocation and Binding

Let $\mathcal{R} = \{R_0, R_1, \dots, R_{w-1}\}$ denote the set of w available computational resources for the PE implementation. We define the following mappings:

- A pipeline rate function $PR : \mathcal{R} \rightarrow \mathbb{N}$
- A mapping $B : \mathcal{S} \rightarrow \mathcal{R}$ which assigns each left hand side variable (and thus each operation) to at least one resource R on which it may be executed

- A function $L : \mathcal{S} \times \mathcal{R} \rightarrow \mathbb{N}$ that assigns a latency to each valid operation/resource pair specified by B .

It is import that B maps each operation to *all* resources that can compute this operation, otherwise the MILP may not find an optimal solution for the binding.

Now, we generate an MILP for scheduling and minimizing the overall latency (Section 3.4). As a result, we obtain a schedule vector λ , an iteration interval P , and a function $\gamma : \mathcal{S} \rightarrow \mathbb{N}$ that assigns a start time within P to each operation. Furthermore, we get a function $B' : \mathcal{S} \rightarrow \mathcal{R}$ which maps each operation to *exactly* one resource R . From this binding, the latency function $L'(s)$ is fixed.

Definition 5.1 Let v_1 and v_2 be indexed variables and $S_{i,j}[I]$ an indexed equation of the form $v_1[I] = f(\dots v_2[I - d^*] \dots)$. We define the statement time displacement \tilde{d}_{v_1, v_2} as

$$\tilde{d}_{v_1, v_2} = d_t^* + \gamma(v_1) - \gamma(v_2) - L'(v_2) \quad (7)$$

with d_t^* as defined in Eq. 5.

Consequently, \tilde{d}_{v_1, v_2} denotes the number of time steps that variable v_2 must be stored after its computation before it is used for the computation of v_1 .

Example 5.2 Consider Equation $S_{2,1}[I]$ of Example 5.1: $u_1[I] = f(y[I - d^*])$ with $d_t^* = 4$. Let $\gamma(u_1) = 2$, $\gamma(y) = 1$, and $L'(y) = 2$ here. So we obtain $\tilde{d}_{u_1, y} = 3$.

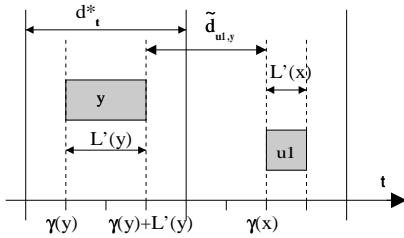


Figure 7: Statement time displacement $\tilde{d}_{u_1, y}$ for variables u_1 and y

5.3 PE Architecture Model

The architecture model of a PE consist of a set of *resource units* (RU), a local *PE controller* (PEC),

and *delay memories*. Each RU computes one or more equations $S_{i,j}[I]$. As shown in Fig. 8, an RU consists of the following components:

- a computational resource that implements the operator functionality,
- input multiplexers that select input data
- output buffers for storing the results until used by other resource units.

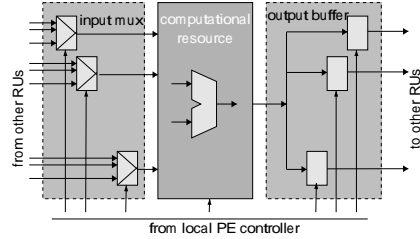


Figure 8: General form of a resource unit.

The output buffer registers are the most area consuming components, so we try to reduce the number of required buffer registers: Results that are not stored for the same time interval can be mapped to the same output buffer, and output buffers can be shared between different RUs, which leads to a RU model slightly different from that shown in Fig. 8.

5.4 Controller Synthesis

We generate a local controller (PEC) for each PE instance which mainly consists of a modulo- P counter and a decoder.

Modulo- P Counter. The PE internal cyclic schedule is controlled by a modulo- P -counter where P is the length of the iteration interval as defined in Section 3.4. The counter state is represented by a $\lceil \log_2 P \rceil$ -bit register.

Control Signal Decoder. The decoder generates the control signal vector from the counter state. The control vector consists of

- select signals for the RU input multiplexers,
- enable signals for the RU output buffers,
- control signals for the computation resources, and
- enable signals for the delay memories (see Section 5.5).

The decoder is a logic network whose function is derived from function γ (Section 5.2). The enable signals for the output buffers are generated whenever a new result must be stored, while control signals for the input multiplexers are generated so that the required data source is selected at the start point $\gamma(S_{i,j}[I])$ of an operation. During the latency interval of an operation, the computation resource control signals are set to the required function (this is only necessary for multi-functional resources). The enable signals for the delay memories are generated when the PE has completed the computation of an indexed variable that is required by another PE at a later time step.

5.5 A New Concept For Efficient FIFO Buffers

The task of the delay memories is to store an indexed variable computed by a PE until it is used by another PE. In general, shift registers or FIFOs are used here. According to Def. 5.1, the length of the shift register is exactly \tilde{d}_{v_1, v_2} . The register is clocked in each clock cycle, resulting in a poor utilization of the register memory since the indexed variable v_2 is computed only once per iteration interval P .

Optimization Of Delay Memories. The number g of data words in the shift register is

$$g = \left\lceil \frac{\tilde{d}_{v_1, v_2}}{P} \right\rceil, \quad (8)$$

so we use shift registers of length g and with a *write_enable*-input, which is activated each time the computation of the indexed variable is completed (that is, once per iteration period). The *write_enable*-input is controlled by the local PEC.

6 Conclusion

We have presented a new methodology for automatically mapping nested loop programs to reconfigurable hardware using our design tool PARO. We have pointed out the requirements that an algorithm must comply with to be suitable for hardware mapping. Then, we have focused on the generation of HDL code for the resulting processor array and presented a way for synthesizing the processor elements as well as the overall array structure.

References

- [1] M. Bednara, O. Beyer, J. Teich, and R. Wanka. Trade-off analysis and architecture design of a hybrid hardware/software sorter. In *ASAP00 - Proc. Int. Conf. on Application-Specific Systems, Architectures, and Processors*, pages 299–308, Boston, U.S.A., July 2000.
- [2] P. Feautrier. Automatic Parallelization in the Polytope Model. Technical Report 8, Laboratoire PRiSM, Université des Versailles St-Quentin en Yvelines, 45, avenue des États-Unis, F-78035 Versailles Cedex, June 1996.
- [3] B. Kienhuis. Matparser: An Array Dataflow Analysis Compiler. Technical report, Department EECS, University of California at Berkeley, Cory Hall 524, Berkeley, California, 94720, USA, Feb. 2000.
- [4] B. Kienhuis, E. Rijpkema, E. Deprettere, and P. Lieverse. High Level Modeling for Parallel Executions of Nested Loop Algorithms. In *IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 79–91, Boston, Massachusetts, 2000.
- [5] R. Kuhn. Transforming algorithms for single-stage and VLSI architectures. *Workshop Interconnection Networks for Parallel and Distributed Processing*, 1980.
- [6] S. Y. Kung. *VLSI Processor Arrays*. Prentice Hall, Englewood Cliffs., 1987.
- [7] D. Moldovan. On the Design of Algorithms for VLSI Systolic Arrays. In *Proceedings of the IEEE*, volume 71, pages 113–120, Jan. 1983.
- [8] R. Schreiber, S. Aditya, B. Rau, V. Kathail, S. Mahlke, S. Abraham, and G. Snider. High-Level Synthesis of Nonprogrammable Hardware Accelerators. In *IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 113–124, Boston, Massachusetts, 2000.
- [9] J. Teich. *A Compiler for Application-Specific Processor Arrays*. PhD thesis, Institut für Mikroelektronik, Universität des Saarlandes, Saarbrücken, Deutschland, September 1993.
- [10] J. Teich and L. Thiele. Control generation in the design of processor arrays. *Int. Journal on VLSI and Signal Processing*, 3(2):77–92, 1991.
- [11] J. Teich and L. Thiele. Partitioning of processor arrays: A piecewise regular approach. *INTEGRATION: The VLSI Journal*, 14(3):297–332, 1993.
- [12] J. Teich, L. Thiele, and L. Zhang. Scheduling of partitioned regular algorithms on processor arrays with constrained resources. In *ASAP96- Proc. Int. Conf. on Application-Specific Systems, Architectures, and Processors*, pages 131–144, Chicago, U.S.A., Aug. 1996.
- [13] 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.
- [14] L. Thiele. Scheduling of uniform algorithms with resource constraints. *Journal of VLSI Signal Processing*, 10:295–310, 1995.