

# Dynamic Piecewise Linear/Regular Algorithms

Frank Hannig and Jürgen Teich

Department of Computer Science 12, Hardware-Software-Co-Design,  
University of Erlangen-Nuremberg, Germany,  
{hannig, teich}@cs.fau.de

## Abstract

*In this paper we present an extension of the class of piecewise linear algorithms (PLAs) in order to model one type of dynamic data dependencies. This extension significantly increases the range of applications which can be parallelized and mapped to massively parallel processor arrays. For instance, a lot of computational intensive applications for video and image processing consist of nested loop programs with only few and simple run-time dependent conditionals. Furthermore, we outline in which case these extensions can directly used – with slight changes – within traditional mapping methodologies based on loop parallelization in the polytope model. Additionally, we outline future research directions in the case existing methods will be inefficient.*

## 1 Introduction

In the last two decades a lot of research has been spent in the area of parallel algorithms that can be systematically mapped onto a class of massive parallel architectures called processor arrays. Today these architectures are of great interest, since progressive integration densities and minimal structures of modern ULSI-devices allow implementations of hundreds of 32-bit microprocessors and more on a single die. Moreover, with the advent of reconfigurable architectures, processor arrays have become flexible as the design of software. Such arrays can solve efficiently a large number of problems in signal, image, and video processing, or numerical linear algebra. Key components of mapping methodologies which can be classified to the area of loop parallelization in the polytope model [5] are linear transformations and schedules in order to derive preferably homogeneous processor arrays with local and regular communication structures and a high degree of pipelining and parallelism. For this purpose, mostly only data flow dominant algorithms with static control have been considered.

Many computational intensive algorithms of the above listed domains have also, in fact only a small and simple control flow which can not be evaluated in advance at compile time but have to be considered at run-time. In order to be able to handle also these algorithms we propose an extension of the class of *piecewise linear algorithms* by one

type of *run-time conditionals*. We describe how these algorithms may be scheduled and mapped by adaptation of existing methods.

## 2 Related Work

Loop parallelization is of great interest in order to accelerate applications either in software or hardware. Transformations can be performed on a program given in an imperative form or in single assignment code (SAC), where the whole parallelism is explicitly expressed. SAC is closely related to a set of recurrence equations, a formalism introduced by Karp, Miller, and Winograd [11]. This formalism has been used in many languages and advanced over the years about affine dependencies or piecewise definitions. E.g., *Systems of Affine Recurrence Equations* (SARE) which are used in the Alpha language [3], the class of *Affine Indexed Algorithms* (AIA) [4], and the class of *Piecewise Linear Algorithms* (PLA) [19, 20]. None of these classes can handle or is used to schedule dynamic data dependencies. As parallelizing compiler, LooPo [6] is mentionable since it cannot only handle static loop bounds like the before described algorithm classes but also while-loops.

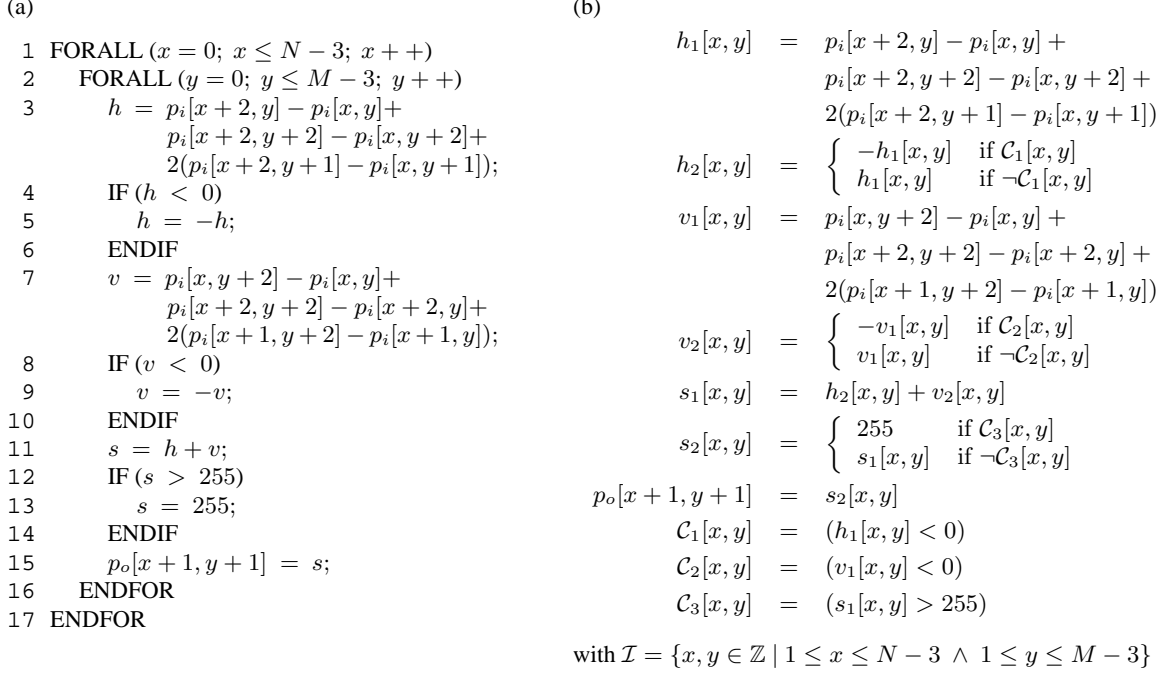
In this area only few synthesis tools for the design of application specific circuits exist: PICO Express [18] which was primarily developed as PICO-N by the Hewlett-Packard Laboratories [12, 17], Compaan [14] which deals with process networks, and PARO [2, 15] which is based on the class of PLAs. PARO is a design system project for modeling, transformation, optimization, and processor synthesis for the class of PLA. PARO can be used during the process of automated synthesis of regular circuits.

## 3 Background and Notation

The purpose of this section is, (i) to recapitulate the class of algorithms we are dealing with called *piecewise linear algorithms* (PLAs), and (ii) to extend this algorithm class by one type of *dynamic data dependencies*.

The class of PLAs has been defined in [19,20]. This class extends the notation of *regular iterative algorithms* [16] that may be related to regular processor arrays. In the following, the properties of PLAs are defined:





**Figure 2.** In (a), pseudo code of an edge detection algorithm. In (b), the same algorithm written in the notation of a DPLA.

else-branch of an if-conditional. We introduce intermediate variables  $x_i^1[s(I)] = \mathcal{F}_i^1(\dots)$  and  $x_i^0[s(I)] = \mathcal{F}_i^0(\dots)$ . Note, the usage of variables defined in one branch is limited to the scope of only this branch, they cannot be used in other parts of the program. A DPLA is called dynamic piecewise regular algorithm (DPRA) if the matrices  $P_i$ ,  $Q_j$ , and  $Q_k$  are the identity matrix.

Note that by this definition we can strictly partition each condition into an iteration dependent conditional and a run-time dependent conditional (separability). Due to both, the run-time dependent conditional ( $\mathcal{C}_i^{\text{RT}}$ ) and the negated run-time dependent conditional ( $\neg\mathcal{C}_i^{\text{RT}}$ ), the left hand side variable of an equation is defined whenever  $\mathcal{C}_i^{\text{I}}(I)$  is fulfilled, and thus the computability property of a program remains satisfied. Furthermore, a corresponding static dependence graph of a DPLA can be specified as will be shown subsequently. But first, in Ex. 3.1 and Ex. 3.2 we give examples of a DPRA and a DPLA, respectively.

### Example 3.1

$$x[i] = \begin{cases} 2 \cdot \frac{\cos(y[i-1])}{\sin(x[i-1])} & \text{if } (x[i-1] \neq 0) \\ \infty & \text{if } (x[i-1] = 0) \end{cases}$$

### Example 3.2

$$x[i, j] = \begin{cases} x^1[i, j] & \text{if } (\mathcal{C}^{\text{RT}}[i, j] \wedge \mathcal{C}^{\text{I}}(I)) \\ x^0[i, j] & \text{if } (\neg\mathcal{C}^{\text{RT}}[i, j] \wedge \mathcal{C}^{\text{I}}(I)) \end{cases}$$

$$\begin{aligned}
x^1[i, j] &= y[i, j] \cdot z[2i - 1, j] \\
x^0[i, j] &= y[i, j] \\
\mathcal{C}^{\text{RT}}[i, j] &= (z[2i - 1, j] > 1) \\
\mathcal{C}^{\text{I}}(I) &= (i > 0 \wedge j \leq 3)
\end{aligned}$$

A PRA might be expressed by a so called *reduced dependence graph* (RDG) [19], also a DPRA can be expressed by a RDG extended by run-time dependent conditionals.

**Definition 3.6 (RCDG).** The reduced control/dependence graph RCDG  $G = (V, E, D)$  associated to a dynamic regular algorithm as defined above is defined as follows: The set of nodes  $V$  can be divided into three disjoint subsets  $V = V_S \cup V_M \cup V_C$ . To each variable  $x_i[I]$  there is associated a simple node  $v_i \in V_S$ . To each equation as defined in Def. 3.5 there are associated two special nodes: (i), one conditional node  $v_{\mathcal{C}_i} \in V_C$  is associated to a run-time dependent conditional  $\mathcal{C}_i^{\text{RT}}$  to determine which of the variables  $x_i^1$ ,  $x_i^0$  is selected in (ii), a merge node  $v_{M_i} \in V_M$ . Since, by definition the nodes  $v_i^1$ ,  $v_i^0$  are only intermediate nodes, they can be grouped together with node  $v_i$  and the corresponding merge node  $v_{M_i}$  to one supernode. In Fig. 1 (a)-(d), graphical symbols of the different node types are shown. Furthermore, a set of edges  $E$  consists of two disjoint subsets  $E = E_D \cup E_C$ . There is an edge  $(v_i, v_j) \in E_D$  with distance vector  $d_{ij}$  if the variable  $x_j$  directly depends on  $x_i$  via the dependence vector  $d_{ij}$ . Each conditional node  $v_{\mathcal{C}_i^{\text{RT}}} \in V_C$  controls one or more merge

nodes  $v_{M_i} \in V_{M_i}$ . These control dependencies are specified by an edge  $(v_{C_i}, v_{M_i}) \in E_C$ .

In the following small example all the definitions are wrapped-up. In the majority of cases, starting point is a given program in a high-level language like C or Java.

**Example 3.3** Consider the following fictive program fragment given in a pseudo language:

```

1  FORALL (i = 1; i ≤ N; i++)
2    FORALL (j = 1; j ≤ M; j++)
3      b[i, j] = b[i, j - 1];
4      a[i, j] = a[i - 1, j] + b[i, j];
5      IF (a[i, j] > 10)
6        a[i, j] = 10;
7      ENDIF
8    ENDFOR
9  ENDFOR

```

This program can be formulated as a DPRA as follows:

$$\begin{aligned}
b[i, j] &= b[i, j - 1] \\
a_1[i, j] &= a_2[i - 1, j] + b[i, j] \\
a_2[i, j] &= \begin{cases} a_2^1[i, j] & \text{if } (a_1[i, j] > 10) \\ a_2^0[i, j] & \text{if } (a_1[i, j] \leq 10) \end{cases} \\
a_2^1[i, j] &= 10 \\
a_2^0[i, j] &= a_1[i, j]
\end{aligned}$$

with the iteration space  $\mathcal{I} = \{i, j \in \mathbb{Z} \mid 1 \leq i \leq N \wedge 1 \leq j \leq M\}$  common to all equations. Note that in order to satisfy the single assignment property, variables have been renamed. This can be performed during a data dependence analysis of a given algorithm [13]. A corresponding reduced dependence graph is depicted in Fig. 1 (e). Here, for each left hand side variable ( $b$ ,  $a_1$ ,  $a_2$ ) of the DPRA exists one node. The evaluation of the run-time dependent conditional is denoted by the triangular-shaped node  $C$  which generates a control signal (dashed edge). This control signal selects in the second triangular-shaped node  $M$  whether the first or the second branch is assigned to variable  $a_2$ .

## 4 Scheduling of DPRAs

Let  $w_C$  be the execution time to evaluate a run-time dependent conditional. Furthermore, let  $w_{\mathcal{F}1}$  and  $w_{\mathcal{F}0}$  be the execution times of the if- and the else-branch of an equation, respectively, and  $w_{\max} = \max\{w_C, w_{\mathcal{F}1}, w_{\mathcal{F}0}\}$ . Then with respect to scheduling dynamic data dependencies, different cases can be considered:

**Nearly balanced branches.** The conditional branches are (nearly) balanced if the following condition holds:  $w_C = w_{\max} \vee w_{\mathcal{F}1} = w_{\mathcal{F}0}$ . Then, two hardware resource models might be considered:

- Assumed enough resources are available, different branches of a run-time dependent conditional may be executed in parallel to achieve highest performance. These types of run-time conditionals are very common in image processing algorithms where often absolute, threshold, or min/max values are computed. Due to the balanced behavior of branches' execution time an optimal static linear schedule can be derived at compile-time.
- If the computation of one branch is more hardware costly, it makes sense to share the resources since different branches of a conditional are mutually exclusive.

**Unbalanced branches.** When the execution times of branches are different (unbalanced,  $|w_{\mathcal{F}1} - w_{\mathcal{F}0}| > 0$ ), linear static scheduling may lead to sub-optimal execution times, since the worst case execution time is always given by the longest branch. Then, worst case and best case run-time estimations might be of interest, or techniques such as loop shifting and compaction as in [7] can be considered in order to balance the branches. If the loop branches may not be balanced properly using branch balancing so that the overhead in execution time would not be tolerable, mixed scheduling concepts consisting of *mixed static/dynamic schedules* or *quasi-static schedules* where events generated at run-time from the evaluation of dynamic data dependencies and trigger statically optimized sub-schedules.

In the following we consider only the case when the branches are simple<sup>1</sup>, nearly balanced and both branches are computed in parallel to allow fastest execution. Then, an optimal static schedule may be derived by the formulation and solving of a *mixed integer linear program* (MILP), similar as in [9, 21]. Therefore, additionally a resource graph has to be specified which expresses the binding possibilities of operations to functional units and execution times and pipeline rates of these units. Due to the sake of brevity and the well-known concepts we omit the MILP formulation here and refer to [9, 21]. The MILP has to satisfy the following conditions: Obviously, one necessary condition to allow the parallel execution is that in the given resource graph there must exist disjoint binding possibilities of operations  $v_C, v_i^1, v_i^0$  for each conditional and its branches. Then the parallel execution is satisfied by the following constraints

1. case,  $w_C = w_{\max}$ :
$$\begin{aligned}
\tau(v_C) &\leq \tau(v_i^1) \leq \tau(v_C) + w_C - w_{\mathcal{F}1} \\
\tau(v_C) &\leq \tau(v_i^0) \leq \tau(v_C) + w_C - w_{\mathcal{F}0}
\end{aligned}$$
2. case,  $w_{\mathcal{F}1} = w_{\mathcal{F}0}$ :

<sup>1</sup>The branches are not nested and can be encapsulated as shown in Fig. 1 (e) and Fig. 3, respectively.

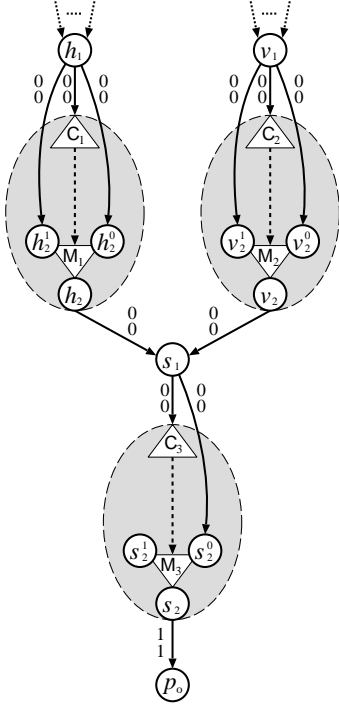


Figure 3. RCDG of the edge detection algorithm.

$$\begin{aligned} \tau(v_i^1) &= \tau(v_i^0) \\ \tau(v_i^1) &\leq \tau(v_c) \leq \tau(v_i^1) + w_{\mathcal{F}1} - w_c \end{aligned}$$

where  $\tau(v_i)$  denotes the relative start time of each operation  $v_i$ .

#### 4.1 Example, Edge Detection

A lot of computational intensive applications for video and image processing consist of nested loop programs with only few and *small* run-time dependent conditionals. As example we consider in the following an edge detection algorithm which is given as pseudo code in Fig. 2. In order to formulate the scheduling problem as a MILP we have to denote the available resources. This can be expressed by a resource graph as shown Fig. 4. An edge in this graph models the possibility that  $v_i$  might be executed on one instance of resource type  $r_k$ . To each edge an execution time of node  $v_i$  on resource type  $r_k$  is associated. Furthermore, to each resource type  $r_k$  a number  $\alpha(r_k)$  of available instances is associated. In the example, the assignment and multiplex operations are considered to have zero clock cycles delay. Furthermore, the parallel execution of branches is possible by the given resource graph. The schedule of one possible and optimal solution is shown in Fig. 5 for three subsequent instances. After a period of  $P = 3$  a new computation can start on the same resources. In Fig. 6, a corresponding hardware realization is depicted.

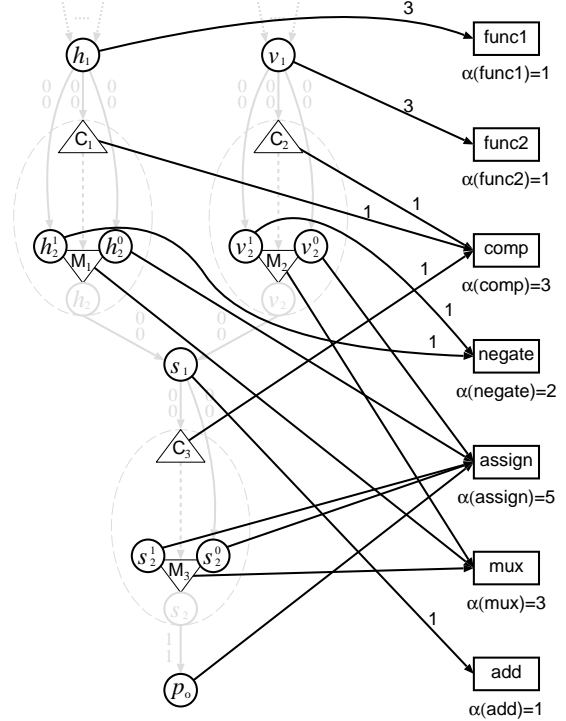


Figure 4. Resource graph of the edge detection algorithm.

## 5 Conclusions and Future Directions

In this paper we presented an extension of the class of PLAs in order to model one type of dynamic data dependencies. This extension significantly increases the range of applications which can be parallelized and mapped to massively parallel processor arrays. Furthermore, we outlined in which case these extensions can directly used – with slight changes – within traditional mapping methodologies based on loop parallelization in the polytope model.

Currently we investigate nested and computational intensive branches where the parallel execution of both branches is too expensive [10]. In the future we would like to consider also unbalanced branches. Here, a two-stage scheduling methodology might be applied: within a branch a static linear schedule can be determined during compile time, around these static parts a dynamic or data flow driven concept has to be developed. In case of reconfiguration at runtime our resource graph has to be extended to allow the modeling of reconfiguration times in order to perform precise worst/best case execution estimations.

The newly class of DPLA introduced in this paper is currently integrated into the PARO design system [1, 15]. Furthermore, in the future we would like to adapt our design methodology in order to target also coarse-grained reconfigurable architectures [8].

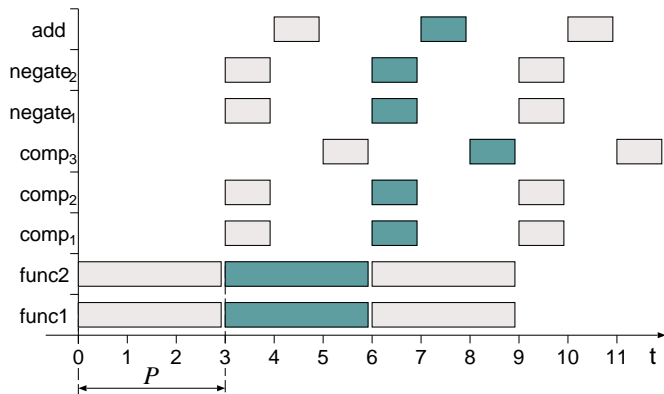


Figure 5. Schedule of the edge detection algorithm.

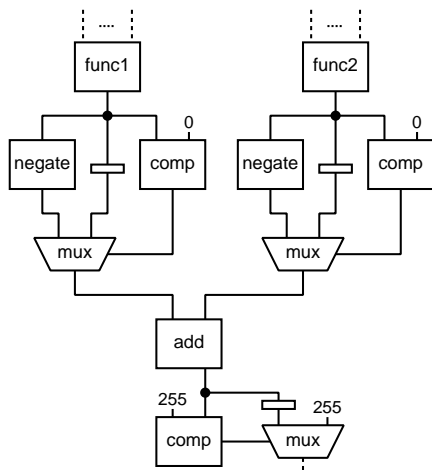


Figure 6. Block diagram of hardware.

## References

- [1] M. Bednara and J. Teich. Synthesis of FPGA Implementations from Loop Algorithms. In *First International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'01)*, pages 1–7, Las Vegas, NV, June 2001.
- [2] M. Bednara and J. Teich. Automatic Synthesis of FPGA Processor Arrays from Loop Algorithms. *The Journal of Supercomputing*, 26(2):149–165, 2003.
- [3] O. S. D. Wilde. Regular Array Synthesis using Alpha. In *Int. Conf. on Application Specific Array Processors, San Francisco, California*, pages 200–211, Aug. 1994.
- [4] 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.
- [5] 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.
- [6] M. Griebel and C. Lengauer. The loop parallelizer LooPo. In M. Gerndt, editor, *Proc. Sixth Workshop on Compilers for Parallel Computers*, volume 21 of *Konferenzen des Forschungszentrums Jülich*, pages 311–320. Forschungszentrum Jülich, 1996.
- [7] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. Loop Shifting and Compaction for the High-Level Synthesis of Designs with Complex Control Flow. In G. Gielen and J. Figueras, editors, *Proceedings of Design, Automation and Test in Europe*, pages 114–119, Paris, France, Feb. 2004. IEEE Computer Society.
- [8] 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)*, Montreal, Quebec, Canada, May 2004.
- [9] 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.
- [10] F. Hannig and J. Teich. Regular Dynamic Dependence Algorithms – Definition and Resource Constrained Scheduling. *Submitted to IEEE 15th International Conference on Application-specific Systems, Architectures and Processors (ASAP 2004)*, Galveston, TX, U.S.A., Sept. 2004.
- [11] R. Karp, R. Miller, and S. Winograd. The Organization of Computations for Uniform Recurrence Equations. *Journal of the Association for Computing Machinery*, 14(3):563–590, 1967.
- [12] V. Kathail, S. Aditya, R. Schreiber, B. R. Rau, D. C. Cronquist, and M. Sivaraman. PICO: Automatically Designing Custom Computers. *Computer*, 35(9):39–47, 2002.
- [13] B. Kienhuis. MatParser: An Array Dataflow Analysis Compiler. Technical Report UCB/ERL M00/9, University of California, Berkeley, CA-94720, U.S.A., Feb. 2000.
- [14] B. Kienhuis, E. Rijpkema, and E. Deprettere. Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures. In *Proceedings of the eighth international workshop on Hardware/software codesign*, pages 13–17. ACM Press, 2000.
- [15] PARO Design System Project. [www12.informatik.uni-erlangen.de/research/paro](http://www12.informatik.uni-erlangen.de/research/paro).
- [16] S. Rao. *Regular Iterative Algorithms and their Implementations on Processor Arrays*. PhD thesis, Stanford University, 1985.
- [17] R. Schreiber, S. Aditya, B. Rau, V. Kathail, S. Mahlke, S. Abraham, and G. Snider. High-Level Synthesis of Non-programmable Hardware Accelerators. Technical Report HPL-2000-31, Hewlett-Packard Laboratories, Palo Alto, May 2000.
- [18] Synfora, Inc. [www.synfora.com](http://www.synfora.com).
- [19] J. Teich. *A Compiler for Application-Specific Processor Arrays*. PhD thesis, Institut für Mikroelektronik, Universität des Saarlandes, Saarbrücken, Deutschland, 1993.
- [20] L. Thiele. *Computer Systems and Software Engineering: State-of-the-Art*, chapter 4, Compiler Techniques for Massive Parallel Architectures, pages 101–151. Kluwer Academic Publishers, Boston, U.S.A., 1992.
- [21] L. Thiele. Resource Constrained Scheduling of Uniform Algorithms. *Journal of VLSI Signal Processing*, 10:295–310, 1995.