Resource Constrained and Speculative Scheduling of Dynamic Piecewise Regular Algorithms

Frank Hannig and Jürgen Teich

{hannig, teich}@cs.fau.de

URL: http://www12.informatik.uni-erlangen.de

Department of Computer Science 12
Hardware-Software-Co-Design
University of Erlangen-Nuremberg
Am Weichselgarten 3
D-91058 Erlangen, Germany

Co-Design-Report 01-2004

June 18, 2004

Contents

Abstract

In this report we present a significant extension of the quantified equation based algorithm class of piecewise regular algorithms. The main contributions of the following report are: (1) the class of piecewise regular algorithms is extended by allowing run-time dependent conditionals, (2) a mixed integer linear program is given to derive optimal schedules of the novel class we call dynamic piecewise regular algorithms, and (3) in order to achieve highest performance, we present a speculative scheduling approach. The results are applied to an illustrative example.

1 Introduction

In the last two decades a lot of research has been done 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 modern nanotechnology 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 [Fea96] 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 cannot 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 in Section 3 an extension of the class of *piecewise regular algorithms* by *run-time dependent conditionals*. In Section 4, we describe how these algorithms can be *scheduled under resource constraints* and how *speculative execution* can be incorporated by adaptation of existing *mixed integer linear programming* methods. Finally in Section 5, we outline future research directions in case existing methods will be inefficient. But first, in the next section we will give a brief overview of related work.

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 given. SAC is closely related to a set of recurrence equations, a formalism introduced by Karp, Miller, and Winograd [KMW67]. 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 [WS94], the class of *Affine Indexed Algorithms* (AIA) [EM99], and the class of *Piecewise Linear Algorithms* (PLA) [Thi92, Tei93]. None of these classes can handle or is used to schedule dynamic data dependencies. As parallelizing compiler, LooPo [GL96] is mentionable since it cannot only handle static loop bounds like the before described algorithm classes but also while-loops.

The authors in [Meg93] and in [AR94] study a class of run-time dependencies such as used in dynamic programming for knapsack problems, i.e., indirect addressing is considered. Further-

more, the authors show how optimal systolic array-like implementations can be derived. In [SD03] the authors present a method to derive so called *dynamic single assignment code* (dSAC) based on *fuzzy array dataflow analysis* [CBF95]. The difference compared to *single assignment code* where every left hand side variable is written exactly once is that in dSAC every variable is written at most once, i.e., at compile-time it is unclear if a variable will be defined or not during the program execution. This is the main difference compared to our approach presented in this report where we assume SAC, i.e., a variable will be defined in either case.

Only few synthesis tools for the design of application specific circuits exist: PICO Express [Syn] which was primarily developed as PICO-N by the Hewlett-Packard Laboratories [SAR+00, KAS+02], Compaan [KRD00] which deals with process networks, and PARO [BT03, PAR] 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. Certainly, there exist a number of fully developed hardware design languages and tools like Handel-C [CEL] or the SPARK environment [GDGN03], but they use imperative forms as input code. Handel-C provides no high-level or parallelizing transformations. The SPARK methodology is particularly targeted to control-intensive applications and incorporates ideas of mutual exclusivity of operations and resource sharing, however, SPARK is restricted to one-dimensional arrays.

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 run-time dependent conditionals.

The class of PLAs has been defined in [Thi92, Tei93]. This class extends the notion of *regular iterative algorithms* [Rao85] that may be related to regular processor arrays.

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

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

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 $\ldots$ 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 *$\mathcal{C}_i^{\mathrm{I}}(I)$ which can equivalently expressed by $I \in \mathcal{I}_{\mathcal{C}_i} \subseteq \mathbb{Z}^n$, where the space $\mathcal{I}_{\mathcal{C}_i}$ 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. Variables that appear on the left hand side of equations are called *defined*. Variables that uniquely appear on the left hand side of equations are called *output variables*. Variables that appear on right hand side of equations are called *used*. Variables that uniquely appear on right hand side of equations are called *input variables*. A program is thus a system of quantified equations that implicitly defines a function of output variables in dependence of input variables. Some other semantical properties are particular to the class of programs we are dealing with.

Single assignment property: Any instance of an indexed variable appears at most once on the left hand side of an equation or, all equations defining the same variable are identical.

Computability: There exists a partial ordering of the equations such that any instance of a variable appearing on the right side of an equation earlier appears in the left hand side in the partial ordering.

Execution Model: The execution model of programs is architecture independent. A program may be executed as follows: (1) All instances of equations are ordered respecting the above defined partial ordering. (2) The indexed variables are determined by successive evaluation of equations.

The domains $\mathcal{I}_i$ are defined as follows:

Definition 3.2 *(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 report, we assume that the matrix M is square and of full rank. Then, each vector κ is uniquely mapped to an index point I. Furthermore, we require that the index space is bounded.

In order to allow not only iteration dependent conditionals which are static and known at compile time, we extend in the following the algorithm class by *run-time dependent conditionals*.

Definition 3.3 *(Run-Time Dependent Conditional). Let $\mathcal{C}^{\mathrm{RT}}[I]$ be a* run-time dependent conditional *of the form*

$$\mathcal{C}^{\mathrm{RT}}[I] \;=\; (\; \mathcal{F}_{\mathcal{C}}(\ldots, y[g(I)], \ldots) \;\; \mathrm{relop} \;\; c \;)$$

where $\mathcal{F}_{\mathcal{C}}(\ldots, x[g(I)], \ldots)$ denotes an arbitrary function involving constants and linearly indexed variables only. $\mathrm{relop} \in \{=, >, \geq, <, \leq, \neq\}$ denotes the set of relational operators and $c \in \mathbb{R}$ is a constant.

Definition 3.4 *(DPLA/DPRA). [HT04] A* dynamic piecewise linear algorithm *(DPLA) is a PLA where an additional type of equations expresses additional restrictions by run-time dependent*

conditionals as follows:

$$\forall I \in \mathcal{I}_i \ : \quad x_i\left[s(I)\right] \ = \ \begin{cases} \mathcal{F}_i^1\left(\ldots, x_j\left[t(I)\right], \ldots\right) & \text{if } \left(\mathcal{C}_i^{\mathrm{I}}(I) \ \wedge \ \mathcal{C}_i^{\mathrm{RT}}\left[I\right]\right) \\ \mathcal{F}_i^0\left(\ldots, x_k\left[u(I)\right], \ldots\right) & \text{if } \left(\mathcal{C}_i^{\mathrm{I}}(I) \ \wedge \ \neg\mathcal{C}_i^{\mathrm{RT}}\left[I\right]\right) \end{cases}$$

with $s(I) = P_i I + f_i$, $t(I) = Q_j I - d_{ji}$, *and* $u(I) = Q_k I - d_{ki}$. *The notation* $\neg\mathcal{C}_i^{\mathrm{RT}}\left[I\right]$ *denotes the negation of the run-time dependent conditional* $\mathcal{C}_i^{\mathrm{RT}}\left[I\right]$, *this is similar to the else-branch of an if-conditional. We introduce intermediate variables* $x_i^1\left[s(I)\right] = \mathcal{F}_i^1\left(\ldots\right)$ *and* $x_i^0\left[s(I)\right] = \mathcal{F}_i^0\left(\ldots\right)$. *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 ($\mathcal{C}_i^{\mathrm{RT}}$) and the negated run-time dependent conditional ($\neg\mathcal{C}_i^{\mathrm{RT}}$), the left hand side variable of an equation is defined whensoever $\mathcal{C}_i^{\mathrm{I}}(I)$ 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

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

Example 3.2

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

A PRA might be expressed by a so called *reduced dependence graph* (RDG) [Tei93], also a DPRA can be expressed by a RDG extended by run-time dependent conditionals.

Definition 3.5 *(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* $V = V_S \cup V_M \cup V_{\mathcal{C}}$. *To each variable* $x_i\left[I\right]$ *there is*

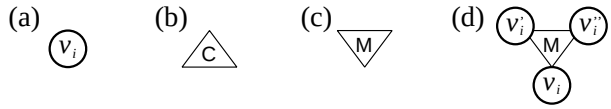

Figure 1: Different node types of a RCDG. (a), simple node, (b), conditional node, (c), merge node, and (d), supernode.

associated a simple node $v_i \in V_S$. *To each equation as defined in Def. 3.4 there are associated two special nodes: (i), one* conditional node $v_{\mathcal{C}_i} \in V_{\mathcal{C}}$ *is associated to a run-time dependent conditional* $\mathcal{C}_i^{\mathrm{RT}}$ *to determine which of the variables* x_i^1, x_i^0 *is selected in (ii), a* merge node $v_{M_i} \in V_{M_i}$. *Since, by definition the nodes* v_i^1, v_i^0 *are only intermediate nodes, they can 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* $E = E_D \cup E_{\mathcal{C}}$. *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* $v_{\mathcal{C}_i^{\mathrm{RT}}} \in V_{\mathcal{C}}$ *controls one or more merge nodes* $v_{M_i} \in V_{M_i}$. *These control dependencies are specified by an edge* $(v_{\mathcal{C}_i}, v_{M_i}) \in E_{\mathcal{C}}$.

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:*

```
FORALL (i = 1; i ≤ N; i++)
   FORALL (j = 1; j ≤ M; j++)
      b[i,j] = b[i,j-1];
      a[i,j] = a[i-1,j] + b[i,j];
      IF (a[i,j] > 10)
         a[i,j] = 10;
      ENDIF
   ENDFOR
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.*

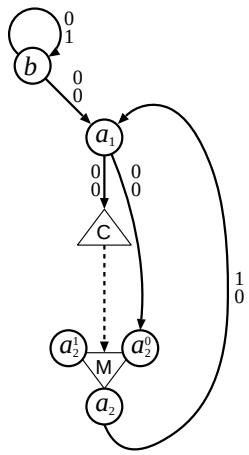

Figure 2: RCDG of the algorithm of Ex. 3.3.

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 similar as in [Kie00] assuming that a variable defined in an if-branch have to be defined also in a corresponding else-branch.

A corresponding reduced dependence graph is depicted in Fig. 2. 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 Allocation and Scheduling of DPRAs

Linear transformations as in the following equation are used as *space-time mappings* [Mol83, Len93] 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 $I \in \mathcal{I}$.

$$\begin{pmatrix} p \\ t \end{pmatrix} = TI = \begin{pmatrix} Q \\ \lambda \end{pmatrix} I \tag{2}$$

In Eq. (2), $Q \in \mathbb{Z}^{(n-1)\times n}$ and $\lambda \in \mathbb{Z}^{1\times n}$. The main reasons for using linear allocation and scheduling functions is that the data flow between processor elements (PE) is local and regular which is essential for VLSI implementations. The interpretation of such a linear transformation is as follows: The set of operations defined at index points $\lambda \cdot I =$ const. are scheduled at the same time step. The index space of allocated processing elements (*processor space*) is denoted by $\mathcal{Q}$ and is given by the set $\mathcal{Q} = \{p \mid p = Q \cdot I \ \wedge \ I \in \mathcal{I}\}$. 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 coprime[1] vector u satisfying $Q \cdot u = 0$ [Kuh80] describes the *processor allocation* equivalently.

Allocation and scheduling must satisfy that no data dependencies in the DG are violated. This is ensured by the well-known *causality constraint*,

$$\lambda \cdot d_{ij} \ \geq \ 0 \quad \forall (v_i, v_j) \in E. \tag{3}$$

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

$$\text{rank} \begin{pmatrix} Q \\ \lambda \end{pmatrix} \ = \ n. \tag{4}$$

Using the projection vector u satisfying $Q \cdot u = 0$, this condition is equivalent to $\lambda \cdot u \neq 0$ [Rao85].

If T is considered to be not anymore square, then among other things also *multi-projections* or *partitioning schemes* like *Local Sequential Global Parallel* (LSGP) partitioning, *Local parallel global sequential* (LPGS) partitioning, or *co-partitioning* [EM99] can be considered by an appropriate decomposition of Q and λ into sub-allocations and sub-schedules, respectively. The description of these methods would exceed the content of this report and would not result in any new insights since the following methods can similarly be applied to them.

In the following we mainly focus on resource constraints in order to derive in practice applicable schedules.

Definition 4.1 *(Resourcegraph). A* resourcegraph $G_R = (V_R, E_R)$ *is a bipartite graph. The set of nodes* $V_R = V \cup V_T$ *contains the node set* V *of the RCDG* $G = (V, E, D)$*. Each node* $r_r \in V_T$ *denotes one* resource type *(e.g., adder, comparator, multiplexer, etc.). An edge* $(v_i, r_k) \in E_R$ *with* $v_i \in V$ *and* $r_k \in V_T$ *models the possibility that* v_i *might be executed on one instance of resource type* r_k*. In case of a supernode only the intermediate nodes* v_i^1*,* v_i^0*, and the node* v_{M_i} *have mapping edges to resource nodes but not the node* v_i*. Furthermore, there exists a weight function* $w : \ E_R \mapsto \mathbb{Z}_0^+$ *which associates to each edge* $(v_i, r_k) \in E_R$ *a time* $w(v_i, r_k)$*, the execution time of* v_i *on* r_k*.*

Let $(G = (V, E, D), G_R = (V_R, E_R))$ be a given specification. An *allocation of functional units* is a function $\alpha : \ V_T \mapsto \mathbb{Z}_0^+$ which associates to each resource type $r_k \in V_T$ a number $\alpha(r_k)$ of available instances. Furthermore, let $\delta(r_k)$ be a pipeline rate which denotes after how many time steps already a new operation can start on r_k.

With this resource model, let $w_{\mathcal{C}}$ 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 $w_{\max} = \max\{w_{\mathcal{C}}, w_{\mathcal{F}^1}, w_{\mathcal{F}^0}\}$. Then with respect to scheduling, different cases can be considered:

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

1. Static parallel branch execution. The conditional branches are (nearly) balanced if the following condition holds: $w_{\mathcal{C}} = w_{\max} \vee w_{\mathcal{F}^1} = w_{\mathcal{F}^0}$. Then, preconditioned 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 [HT04].

2. Static mutually exclusive scheduling. 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.

3. Branch Balancing. 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, techniques such as loop shifting and compaction as in [GDGN04] can be investigated in order to balance the branches.

4. Quasi-static scheduling. If 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 run-time dependent conditionals and trigger statically optimized sub-schedules.

In the following we present a methodology, for the second case, to schedule DPRAs under resource constraints.

4.1 Static mutually exclusive scheduling

An entire mixed integer linear program (MILP) to derive optimal schedules is given in Appendix A. In this section we want to consider only the novel formulations in order to account mutually exclusive operations with resource constraints. If the number of available resources is limited, several operations may compete for the same resource. Then it has to be prevented that more than $\alpha(r_k)$ operations are being simultaneously executed by the same resource type $r_k \in V_T$. Here, we have to distinguish two different cases

1. Concurrent operations, we say also the operations are in AND relation,
2. If there are different execution branches in an algorithm, the operations are mutually exclusive, in XOR relation.

The relationships among a set of operations can be represented as a tree where the internal nodes are of the above introduced types, XOR and AND, and the leaves are the operations [HHL90]. Let a node of the relationship tree have n sub-trees and the number of possible concurrent operations for each sub-tree be $f_j(r_k, t)$, $j = 1 \ldots n$. With this, the resource constraints can be formulated as

follows:

$$f_j(r_k,t) \;\leq\; \alpha(r_k) \qquad \begin{array}{l} \forall\, r_k \in V_T \\ \forall\, t : 0 \leq t \leq P-1 \end{array} \tag{5}$$

where, $f_j(r_k,t)$ is defined as follows[2,3]

$$f_j(r_k,t) = \begin{cases} \sum_{j=1}^{n} f_j(r_k,t) & \text{if the node is an AND node} \\ \max_{j=1}^{n} f_j(r_k,t) & \text{if the node is an XOR node} \\ \sum_{d=0}^{\delta(r_k)-1} \sum_{\forall \nu : l_i \leq t-d-\nu P \leq h_i} x_{i,k,t-d-\nu P} & \text{if the node is a leaf } v_i \;\wedge\; \exists\,(v_i,r_k) \in E_R \\ 0 & \text{if the node is a leaf } v_i \;\wedge\; \nexists\,(v_i,r_k) \in E_R \end{cases}$$

The constraint in Eq. (5) has to be generated for all different resource types and for all minus one time steps in the iteration interval. If one leaf node v_i with binding possibility to resource node r_k is reached, a sum of binary variables is generated. Here, by the outer summation the pipeline rate of resource type r_k is considered. During the time steps from 0 to $\delta(r_k)-1$ only one operation can be executed on one instance of resource type r_k. Since an operation is executed repeatedly, i.e., with a distance of P time instances, operations in different iterations may overlap and multiplies of the iteration interval (νP) have to be considered. This is ensured by the innermost summation where the condition $l_i \leq t-d-\nu P \leq h_i$ can be interpreted as a *convolution* of the scheduling intervals to the basic iteration interval. l_i and h_i are lower and upper bounds of t, for further details see Appendix A.

In a linear program the above inequalities can be formulated only for a fixed iteration interval P (for Def., see Footnote 4) but as can seen in Appendix A, P is a variable of the MILP. This problem can be solved by the introduction of an upper bound P_{max} and the introduction of further binary variables for all possible values of P. We omit this procedure for the sake of brevity.

4.2 Example

In this section, the proposed MILP formulation that can be used to schedule dynamic piecewise regular algorithms under resource constraints and to allocate necessary resources simultaneously is applied to an illustrative example. For this, consider the fictive nested loop program given in Fig. 3 (a). In this small program two nested run-time conditionals are present. An equivalent DPRA of the algorithm is represented in Fig. 3 (b).

In order to formulate the scheduling problem as a MILP we have to denote the available resources. Therefore, in Fig. 4 (a) the RCDG of the algorithm and in Fig. 4 (b) a resource graph is

[2] $x_{i,k,t}$ is a binary variable of the MILP denoting, if true, that the execution of operation v_i on resource type r_k is started at time t. For further details of the MILP see Appendix A.

[3] A term $b = \max(a_1, ..., a_n)$ can be calculated in a minimization problem by n inequalities $b \geq a_i$, $i = 1 \ldots n$ and the addition of b to the objective function, $f(x) + b$.

(a)

```
FORALL (i = 1; i ≤ N; i++)
    FORALL (j = 1; j ≤ M; j++)
        b[i,j] = b[i,j−1] − c[i−1,j];
        a[i,j] = a[i−1,j] + b[i,j];
        IF (a[i,j] > 10)
            c[i,j] = b[i,j] · b[i,j];
            IF (b[i,j] > 8)
                d[i,j] = b[i,j] + a[i,j];
                e[i,j] = a[i,j] + 10;
            ELSE
                d[i,j] = 2 · b[i,j];
                e[i,j] = a[i,j] + 3;
            ENDIF
        ELSE
            c[i,j] = a[i,j] + 8;
            d[i,j] = b[i,j] + 3;
            e[i,j] = 2 · a[i,j];
        ENDIF
    ENDFOR
ENDFOR
```

(b)

$$
\begin{aligned}
a[i,j] &= a[i-1,j] + b[i,j] \\
b[i,j] &= b[i,j-1] - c[i-1,j] \\
c[i,j] &= \begin{cases} c^1[i,j] & \text{if } \mathcal{C}_1[i,j] \\ c^0[i,j] & \text{if } \neg\mathcal{C}_1[i,j] \end{cases} \\
c^1[i,j] &= b[i,j] \cdot b[i,j] \\
c^0[i,j] &= a[i,j] + 8 \\
d[i,j] &= \begin{cases} d^1[i,j] & \text{if } \mathcal{C}_1[i,j] \\ d^0[i,j] & \text{if } \neg\mathcal{C}_1[i,j] \end{cases} \\
d^1[i,j] &= \begin{cases} d^{1.1}[i,j] & \text{if } \mathcal{C}_2[i,j] \\ d^{1.0}[i,j] & \text{if } \neg\mathcal{C}_2[i,j] \end{cases} \\
d^0[i,j] &= b[i,j] + 3 \\
d^{1.1}[i,j] &= b[i,j] + a[i,j] \\
d^{1.0}[i,j] &= 2 \cdot b[i,j] \\
e[i,j] &= \begin{cases} e^1[i,j] & \text{if } \mathcal{C}_1[i,j] \\ e^0[i,j] & \text{if } \neg\mathcal{C}_1[i,j] \end{cases} \\
e^1[i,j] &= \begin{cases} e^{1.1}[i,j] & \text{if } \mathcal{C}_2[i,j] \\ e^{1.0}[i,j] & \text{if } \neg\mathcal{C}_2[i,j] \end{cases} \\
e^0[i,j] &= 2 \cdot a[i,j] \\
e^{1.1}[i,j] &= a[i,j] + 10 \\
e^{1.0}[i,j] &= a[i,j] + 3 \\
\mathcal{C}_1[i,j] &= (a[i,j] > 10) \\
\mathcal{C}_2[i,j] &= (b[i,j] > 8) \quad \text{if } \mathcal{C}_1[i,j]
\end{aligned}
$$

with $\mathcal{I} = \{i, j \in \mathbb{Z} \mid 1 \leq i \leq N \ \wedge \ 1 \leq j \leq M\}$

Figure 3: In (a), pseudo code of a nested loop program, in (b), the same algorithm written as a DPRA.

shown. Here, the available resources consist of one subtracter (sub), one adder (add), two comparators (cmp), and one multiplier (mul). Thus, six add operations and three mul operations have to share each only one functional unit. The possibility of *module selection* can be seen for node b which can be mapped either on resource type sub or on resource type add. All resources require one clock cycle except the multiplier which requires two cycles, but due to pipelining also the multiplier is able to start every δ(mul) = 1 clock cycle a new operation. Note, we assume that the multiplexers M_1 to M_5 have zero time overhead, hence binding possibilities and multiplexer resources are not depicted in the resource graph.

The relationship tree of the program is shown in Fig. 5 (a), accordingly to the nesting level of the if-conditionals the tree has levels. With this information and a given allocation by a projection vector $u = (1\ \ 0)^{\mathrm{T}}$ the minimization problem can be formulated. As a valid and optimal solution we obtain as schedule vector $\lambda = (5\ \ 1)$, as iteration interval $P = 5$, and $\tau(\mathcal{C}_1) = 2$, $\tau(\mathcal{C}_2) = 3$, $\tau(b) = 0$, $\tau(a) = 1$, $\tau(c^1) = 3$, $\tau(c^0) = 3$, $\tau(d^0) = 4$, $\tau(e^0) = 3$, $\tau(d^{1.1}) = 5$, $\tau(e^{1.1}) = 4$, $\tau(d^{1.0}) = 4$, $\tau(e^{1.0}) = 4$ are the relative start times for each operation. In Fig. 6 the schedule is visualized for two processor elements for a time of each three periods. Only some data dependencies are shown. For processor element one, from left to right, three different schedules for the cases $\mathcal{C}_1[i,j] = true \wedge \mathcal{C}_2[i,j] = true$, $\mathcal{C}_1[i,j] = true \wedge \mathcal{C}_2[i,j] = false$, and $\mathcal{C}_1[i,j] = false$ are shown. Furthermore, it can be seen that the execution of different iterations overlap within the iteration interval P. In terms of power awareness the schedule is conscious but in terms of resource utilization, the adder and multiplier are working only 50% of the time. Therefore, it can be considered to use the free resources in order to achieve a better performance. Here, the main idea is to precompute some variables in advance before the run-time conditional is evaluated. This strategy, called *speculative execution* is well known and used in super-scalar and VLIW processors [SBV98]. This method will be formulated in the next section for the first time in the realm of processor array design.

4.3 Speculative Scheduling

In order to achieve highest performance, free resources can be used to precalculate speculative some variables. The main ideas to formulate the speculative execution of operations within our MILP can be summarized as follows:

- Let $\tau(\mathcal{C}_i^{\mathrm{RT}})$ be the fixed execution start times of the run-time dependent conditionals.
- Then, for each run-time dependent conditional two cases can be considered:
  1. $t < \tau(\mathcal{C}_i^{\mathrm{RT}}) + w_{\mathcal{C}_i^{\mathrm{RT}}}$: The conditional has not been yet computed. Thus all of its depending nodes are in AND relation.
  2. $t \geq \tau(\mathcal{C}_i^{\mathrm{RT}}) + w_{\mathcal{C}_i^{\mathrm{RT}}}$: The conditional has been evaluated and thus its branches are in XOR relation in the following time.

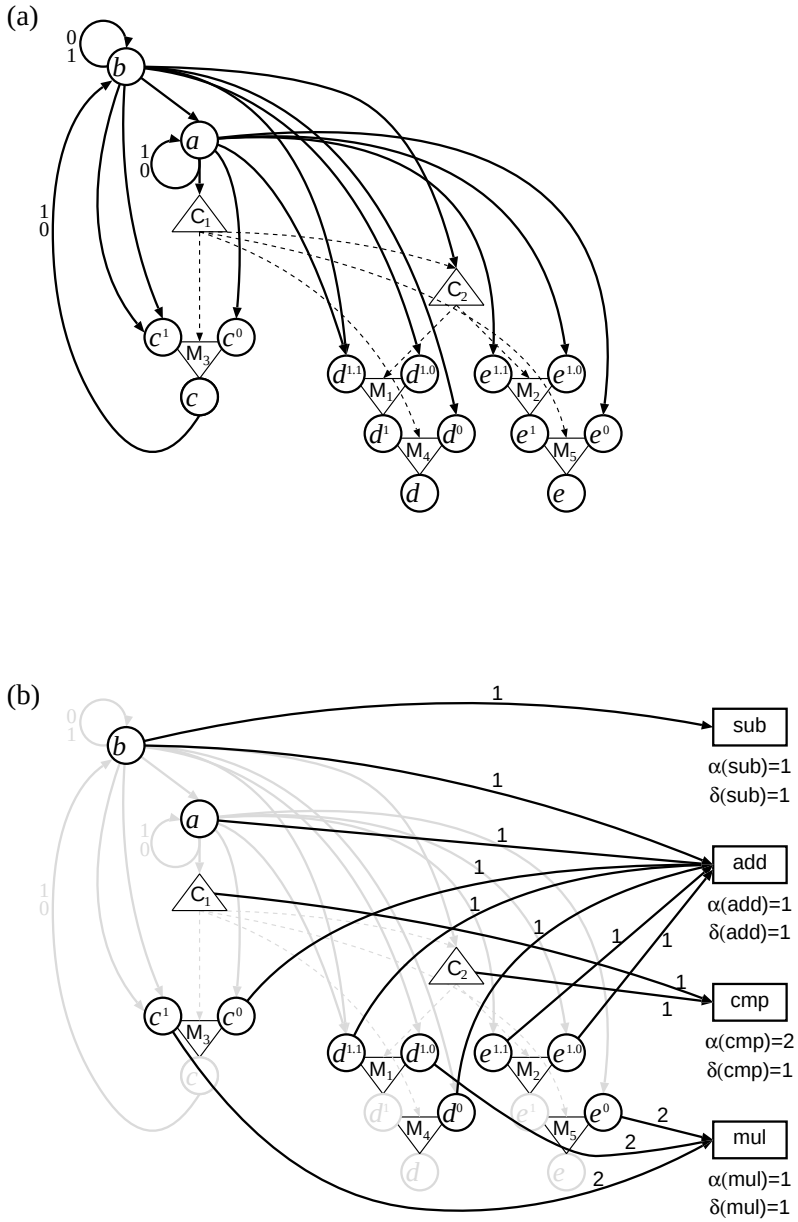

Figure 4: The RCDG of the algorithm in Fig. 3 (b) is shown in (a) and a corresponding resource graph in (b), respectively.

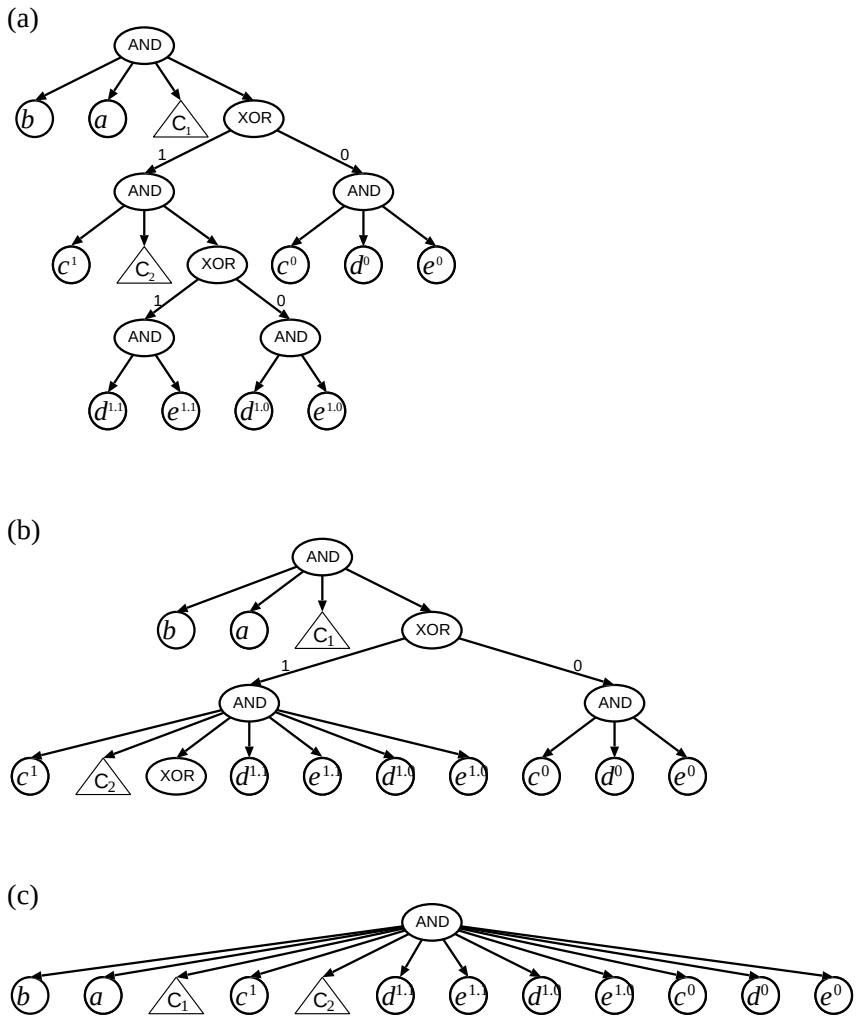

Figure 5: In (a), relationship tree of the algorithm in Fig. 3. In (b) and (c), relationship trees in dependence on time intervals in order to perform speculation.

- Furthermore, if multi-cycle operations are allowed it must be ensured that the operations' execution can be interrupted by others.

For the before discussed algorithm in Fig. 3 we get three intervals:

$$t < \tau(\mathcal{C}_1) + w_{\mathcal{C}_1} \tag{6}$$

$$\tau(\mathcal{C}_1) + w_{\mathcal{C}_1} \leq t < \tau(\mathcal{C}_2) + w_{\mathcal{C}_2} \tag{7}$$

$$\tau(\mathcal{C}_2) + w_{\mathcal{C}_2} \leq t \tag{8}$$

In the last interval (Eq. (8)) all run-time dependent conditionals have been evaluated such that this case is equivalent to the relationship tree in Fig. 5 (a). The first (Eq. (6)) and the second (Eq. (7)) interval are depicted in Fig. 5 (c) and (b), respectively. Formulating the MILP with these relationship trees, leads to the optimized schedule shown in Fig. 7 with the schedule vector

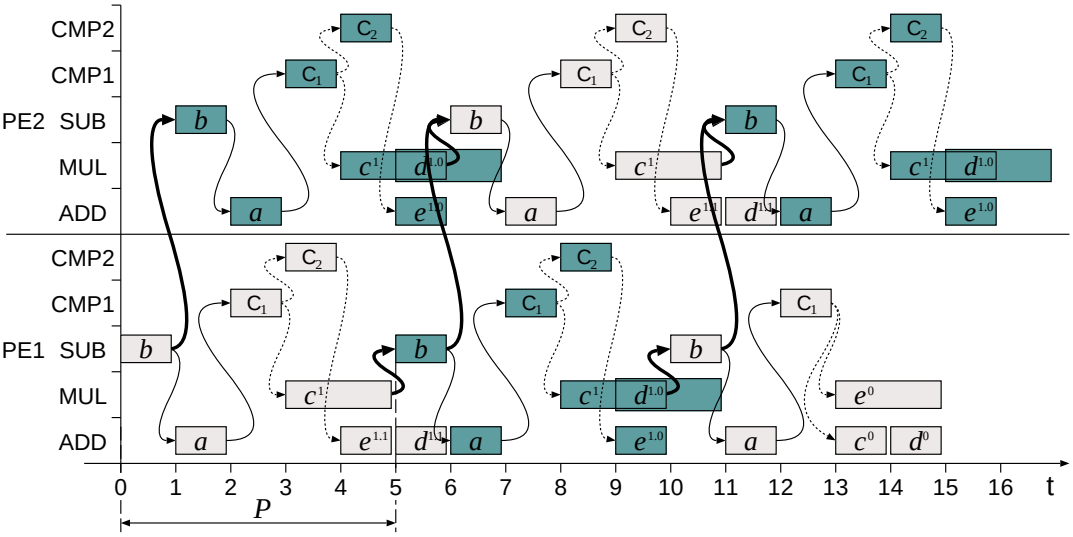

Figure 6: Gantt chart for a part of the schedule. Depicted are two processor elements for a time of each three periods.

$\lambda = (4\ \ 1)$, as iteration interval $P = 4$, and $\tau(\mathcal{C}_1) = 2$, $\tau(\mathcal{C}_2) = 3$, $\tau(b) = 0$, $\tau(a) = 1$, $\tau(c^1) = 1$, $\tau(c^0) = 2$, $\tau(d^0) = 3$, $\tau(e^0) = 3$, $\tau(d^{1.1}) = 3$, $\tau(e^{1.1}) = 4$, $\tau(d^{1.0}) = 2$, $\tau(e^{1.0}) = 4$. For instance, variables c^1 and c^0 are calculated speculative, in advance and in parallel to $\mathcal{C}_1$. After the comparison, also c^1 and c^0 have been already computed, one variable is selected for the further dataflow and the other is disallowed (striped bar in the Gantt chart).

5 Conclusions and Future Directions

In this report we presented an extension of the class of PRAs in order to model run-time dependent conditionals. 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 *small* run-time dependent conditionals. Furthermore, we presented an exact scheduling methodology which takes resource constraints and mutual exclusive execution paths into account. Finally, we presented novel extensions when designing processor arrays to utilize not used resources in order to perform speculative computations to achieve better execution performance.

In the future we would like to investigate unbalanced and computational intensive 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 run-time 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 report is currently integrated into the PARO design

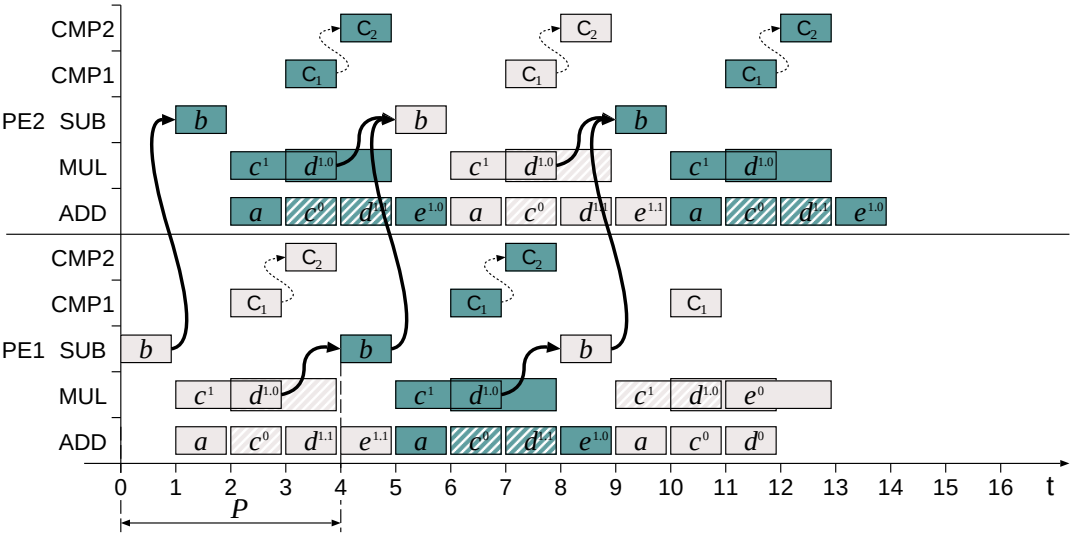

Figure 7: Gantt chart for a part of the speculative schedule.

system [PAR, BT01]. Furthermore, in the future we would like to adapt our design methodology in order to target also coarse-grained reconfigurable architectures [HDT04].

References

[AR94] Rumen Andonov and Sanjay Rajopadhye. An Optimal Algo-Tech-Cuit for the Knapsack Problem. Technical Report PI-791, IRISA, Campus de Beaulieu, Rennes, France, January 1994.

[BT01] Marcus Bednara and Jürgen 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.

[BT03] Marcus Bednara and Jürgen Teich. Automatic Synthesis of FPGA Processor Arrays from Loop Algorithms. *The Journal of Supercomputing*, 26(2):149–165, 2003.

[CBF95] Jean-François Collard, Denis Barthou, and Paul Feautrier. Fuzzy Array Dataflow Analysis. In *Proceedings of the fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 92–101. ACM Press, 1995.

[CEL] CELOXICA, Handel-C. `www.celoxica.com`.

[DR92] A. Darte and Y. Robert. Scheduling uniform loop nests. Technical Report 92–10, ENSL Lyon, France, 1992.

[EM99] Uwe Eckhardt and Renate 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.

[Fea96] Paul 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.

[GDGN03] S. Gupta, N. D. Dutt, R. K. Gupta, and A. Nicolau. SPARK: A High-Level Synthesis Framework for Applying Parallelizing Compiler Transformations. In *Proceedings of the International Conference on VLSI Design*, January 2003.

[GDGN04] Sumit Gupta, Nikil Dutt, Rajesh Gupta, and Alexandru Nicolau. Loop Shifting and Compaction for the High-Level Synthesis of Designs with Complex Control Flow. In Georges Gielen and Joan Figueras, editors, *Proceedings of Design, Automation and Test in Europe*, pages 114–119, Paris, France, February 2004. IEEE Computer Society.

[GL96] Martin Griebl and Christian Lengauer. The Loop Parallelizer LooPo. In Michael Gerndt, editor, *Proc. Sixth Workshop on Compilers for Parallel Computers*, volume 21 of *Konferenzen des Forschungszentrums Jülich*, pages 311–320. Forschungszentrum Jülich, 1996.

[HDT04] Frank Hannig, Hritam Dutta, and Jürgen Teich. Regular Mapping for Coarse-grained Reconfigurable Architectures. In *Proceedings of the 2004 IEEE International Conference on Acoustics, Speech, and Signal Pro cessing (ICASSP 2004)*, volume V, pages 57–60, Montréal, Quebec, Canada, May 2004. IEEE Signal Processing Society.

[HHL90] Cheng-Tsung Hwang, Yu-Chin Hsu, and Youn-Long Lin. Optimum and Heuristic Data Path Scheduling under Resource Constraints. In *Conference proceedings on 27th ACM/IEEE design automation conference*, pages 65–70. ACM Press, 1990.

[HT01] Frank Hannig and Jürgen Teich. Design Space Exploration for Massively Parallel Processor Arrays. In Victor Malyshkin, editor, *Parallel Computing Technologies, 6th International Conference, PaCT 2001, Proceedings*, volume 2127 of *Lecture Notes in Computer Science (LNCS)*, pages 51–65, Novosibirsk, Russia, September 2001. Springer.

[HT04] Frank Hannig and Jürgen Teich. Dynamic Piecewise Linear/Regular Algorithms. In *Proceedings of the Fourth International Conference on Parallel Computing in Electrical Engineering (PARELEC 2004)*, Dresden, Germany, September 2004.

[KAS+02] Vinod Kathail, Shail Aditya, Robert Schreiber, B. Ramakrishna Rau, Darren C. Cronquist, and Mukund Sivaraman. PICO: Automatically Designing Custom Computers. *Computer*, 35(9):39–47, 2002.

[Kie00] Bart Kienhuis. MatParser: An Array Dataflow Analysis Compiler. Technical Report UCB/ERL M00/9, University of California, Berkeley, CA-94720, U.S.A., February 2000.

[KMW67] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The Organization of Computations for Uniform Recurrence Equations. *Journal of the Association for Computing Machinery*, 14(3):563–590, 1967.

[KRD00] Bart Kienhuis, Edwin Rijpkema, and Ed F. Deprettere. Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures. In *Proceedings of the 8th International Workshop on Hardware/Software Co-Design*, pages 13–17. ACM Press, 2000.

[Kuh80] Robert H. Kuhn. Transforming Algorithms for Single-Stage and VLSI Architectures. In *Workshop on Interconnection Networks for Parallel and Distributed Processing*, pages 11–19, West Layfaette, IN, April 1980.

[Len93] Christian Lengauer. Loop Parallelization in the Polytope Model. In Eike Best, editor, *CONCUR'93*, Lecture Notes in Computer Science 715, pages 398–416. Springer-Verlag, 1993.

[Meg93] Graham M. Megson. Mapping a Class of Run-Time Dependencies onto Regular Arrays. In *Proceedings 7th International Parallel Processing Symposium*, pages 97–104, Newport Beach, CA, April 1993. IEEE Computer Society Press.

[Mol83] Dan I. Moldovan. On the Design of Algorithms for VLSI Systolic Arrays. In *Proceedings of the IEEE*, volume 71, pages 113–120, January 1983.

[PAR] PARO Design System Project. `www12.informatik.uni-erlangen.de/research/paro`.

[Rao85] S. K. Rao. *Regular Iterative Algorithms and their Implementations on Processor Arrays*. PhD thesis, Stanford University, 1985.

[SAR+00] Robert Schreiber, Shail Aditya, B. Ramakrishna Rau, Vinod Kathail, Scott Mahlke, Santosh Abraham, and Greg Snider. High-Level Synthesis of Nonprogrammable Hardware Accelerators. Technical Report HPL-2000-31, Hewlett-Packard Laboratories, Palo Alto, May 2000.

[SBV98] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *25 Years of the International Symposia on Computer Architecture (selected papers)*, pages 521–532. ACM Press, 1998.

[Sch86] Alexander Schrijver. *Theory of Linear and Integer Programming*. Whily – Interscience series in discrete mathematics. John Wiley & Sons, Chichester, New York, 1986.

[SD03] Todor Stefanov and Ed F. Deprettere. Deriving Process Networks from Weakly Dynamic Applications in System-Level Design. In *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign & System synthesis (CODES+ISSS'03)*, pages 90–96. ACM Press, 2003.

[Syn] Synfora, Inc. `www.synfora.com`.

[Tei93] Jürgen Teich. *A Compiler for Application-Specific Processor Arrays*. PhD thesis, Institut für Mikroelektronik, Universität des Saarlandes, Saarbrücken, Deutschland, 1993.

[Thi92] Lothar 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.

[Thi95] Lothar Thiele. Resource Constrained Scheduling of Uniform Algorithms. *Journal of VLSI Signal Processing*, 10:295–310, 1995.

[TTZ97] Jürgen Teich, Lothar Thiele, and Li Zhang. Scheduling of Partitioned Regular Algorithms on Processor Arrays with Constrained Resources. *Journal of VLSI Signal Processing*, 17(1):5–20, September 1997.

[WS94] Doran Wilde and Oumarou Sié. Regular Array Synthesis using Alpha. In *Int. Conf. on Application Specific Array Processors, San Francsico, California*, pages 200–211, August 1994.

A MILP Formulation

In this section the construction of a MILP is described. Its solution yields a latency optimal schedule and resource type binding of operations. This methodology mainly recapitulates previous ideas reported in [Thi95, TTZ97, HT01].

As is typical for VLSI processor array design, a *linear schedule* is applied to the DPRA with the index space $\mathcal{I}$. The calculation of a variable (or an operation) $x_i\,[I]$ for $I \in \mathcal{I}$ starts at time step

$$t(I) = \lambda \cdot I + \tau(v_i) \tag{9}$$

and is finished at

$$t(I) = \lambda \cdot I + \tau(v_i) + w_i, \tag{10}$$

respectively.

A.1 Objective Function

The goal of the optimization is to minimize the total evaluation time L which is given by:

$$L \;=\; \max_{I \in \mathcal{I}} \left\{ \lambda \cdot I + \max_{v_i \in V} \{\tau(v_i) + w_i\} \right\} - \min_{I \in \mathcal{I}} \left\{ \lambda \cdot I + \min_{v_i \in V} \{\tau(v_i)\} \right\} \tag{11}$$

In the following, L is approximated by

$$\begin{array}{rl} \max & \lambda \cdot (I_2 - I_1) \\ \text{subject to} & A \cdot I_1 \geq b \\ & A \cdot I_2 \geq b \end{array}$$

Using the duality theorem of linear programming [Sch86, DR92] the problem can be rewritten as:

$$\begin{array}{rll} \min & -(y_1 + y_2) \cdot b & \\ \text{subject to} & y_1 \cdot A \;=\; \lambda & y_1 \;\geq\; 0\,,\; y_1 \in \mathbb{Q}^{1 \times m} \\ & y_2 \cdot A \;=\; -\lambda & y_2 \;\geq\; 0\,,\; y_2 \in \mathbb{Q}^{1 \times m} \\ & \lambda \in \mathbb{Z}^{1 \times n} & \end{array}$$

A.2 Dependence Constraints

The implied data dependencies given in the RCDG have to be guaranteed. I.e., for all edges $(v_i, v_j) \in E$, the computation of node v_j can not begin until the execution of node v_i is finished. Therefore, the starting time of v_j must be at least w_i greater than the starting time of node v_i. The corresponding constraint

$$t_j(I) \;-\; t_i(I - d_{ij}) \;\;\geq\;\; w_i \;\forall\, (v_i, v_j) \in E$$

directly leads with Eq. (9) to

$$\lambda \cdot d_{ij} \;+\; \tau\,(v_j) - \tau\,(v_i) \;\;\geq\;\; w_i$$

A.3 Determination of the Iteration Interval

The *iteration interval*[4] is given by $P = |\lambda \cdot u|$. Since the iteration interval is not a fixed parameter it has to be considered as variable P in the linear program. Let P_{max} be an upper bound of the iteration interval, then the absolute value $|\lambda \cdot u|$ can be determined by the following set of inequalities:

[4] The iteration interval P of an allocated and scheduled piecewise regular algorithm is the number of time instances between the evaluation of two successive instances of a variable within one processing element [Thi95].

$$
\begin{aligned}
P &\leq P_{max} & P \in \mathbb{Z} \\
\lambda \cdot u &\geq P - 2vP_{max} & v \in \{0,1\} \\
\lambda \cdot u &\leq P \\
-\lambda \cdot u &\geq P - 2(1-v)P_{max} \\
-\lambda \cdot u &\leq P
\end{aligned} \tag{12}
$$

It can easily demonstrated that the binary variable v always takes the *right value* so that a positive value follows, i.e., the absolute value of P.

A.4 Resource Constraints

In order to take into account resource constraints within a processing element, an additional representation is introduced. Where, the main concepts can be summarized as follows:

- The scheduling of operation $v_i \in V$ is characterized by a binary variable $x_{i,k,t}$, where $x_{i,k,t} = 1$ denotes that at relative time instance $t \in \mathbb{Z}$ the operation v_i starts its operation on resource type $r_k \in V_T$.

- The relative start time $\tau(v_i)$ of each operation in the RCDG has a lower and upper bound. These bounds can be obtained from schedules following the well known *as soon as possible* (ASAP) and *as late as possible* (ALAP) principles, respectively. For this, within the processor elements unlimited resources are assumed. Through this, for each operation $v_i \in V$ an interval of possible starting times is determined, $\tau(v_i) \in [l_i, h_i]$, where l_i denotes the earliest and h_i the latest possible starting time, respectively.

- The relative start times $\tau(v_i)$ in Eq. (9) are represented as the weighted sum of the binary variables $x_{i,k,t}$

$$\tau(v_i) = \sum_{\forall k:(v_i,r_k)\in E_R} \sum_{t=l_i}^{h_i} t\, x_{i,k,t} \quad \forall\, v_i \in V$$

- As each operation can be executed on one resource type only, we need to add the following equalities

$$\sum_{\forall k:(v_i,r_k)\in E_R} \sum_{t=l_i}^{h_i} x_{i,k,t} = 1 \quad \forall\, v_i \in V$$

- The evaluation time corresponding to a node $v_i \in V$ executed on a resource type $r_k \in V_T$ can then be determined as

$$w_i = \sum_{\forall k:(v_i,r_k)\in E_R} \sum_{t=l_i}^{h_i} w(v_i, r_k)\, x_{i,k,t} \quad \forall\, v_i \in V$$

- And finally, in order to model resource constraints of mutually exclusive operations, the constraints that have been defined in Eq. (5) are used.