VLSI Design
1995, Vol. 3, No. 1, pp. 67–80
Reprints available directly from the publisher
Photocopying permitted by license only

Stepwise Transformation of Algorithms into Array Processor Architectures by the DECOMP

UWE VEHLIES
Laboratorium für Informationstechnologie, University of Hannover
Schneiderberg 32, 30167 Hannover 1, FRG

(Received May 12, 1993, Revised January 4, 1994)

A formal approach for the transformation of computation intensive digital signal processing algorithms into suitable array processor architectures is presented. It covers the complete design flow from algorithmic specifications in a high-level programming language to architecture descriptions in a hardware description language. The transformation itself is divided into manageable design steps and implemented in the CAD-tool DECOMP which allows the exploration of different architectures in a short time. With the presented approach data independent algorithms can be mapped onto array processor architectures. To allow this, a known mapping methodology for array processor design is extended to handle inhomogeneous dependence graphs with nonregular data dependences. The implementation of the formal approach in the DECOMP is an important step towards design automation for massively parallel systems.

Key Words: *Computer-Aided Synthesis; Synthesis Algorithms; CAD for Architecture Design; Stepwise Transformation; Mapping Methodology; Data Independent Algorithms; Run Time Protocol; Nonregular Dependence Graphs; Array-Processor Architectures*

1. INTRODUCTION

Progress in VLSI technology allows to integrate more and more transistors into a single chip. Thus, microelectronic systems with an increasing complexity can be realized. But this also results in a large quantity of design work manageable only with efficient support by design tools.

At the same time the algorithms developed in digital signal processing (DSP) grow in their complexity thereby requiring more and more computational power and higher throughput rates. This is in particular the case in the area of image and video processing where algorithms for high definition television (HDTV) and video telephone have to be applied under real time conditions. Application specific integrated circuits (ASICs) for such systems can be realized only using special purpose architectures (cf [1]). One possible architecture are array processors [2] because they meet the requirements by a massive application of pipelining and parallel processing. In addition, due to their regularity and modularity array processors are well suited for a design process automated by design tools. These trends influence the design methodology for microelectronic systems. In the past design work mainly consists of logic design and layout synthesis. Today these design tasks are well supported by commercial tools. But there is a need to extend these tools because increasing emphasis is given to decisions at the architecture level.

Today the derivation of architectures is manually performed by an intensive and error prone process. In most cases only few different architectures are examined. Due to this an unsuitable architecture may be derived and it becomes impossible to fulfill the requirements of a given algorithm. Thus, design methodologies supporting the architecture level must be developed and implemented in CAD-tools which enable designers to explore different architectures in a short time.

In this direction a lot of research is performed. But due to the complexity of the design process the solutions inevitably are restricted to small and/or

regular design problems. Exploiting regularity several methodologies for mapping algorithms onto architectures have been developed [2, 3, 4, 5, 6, 7, 8, 9, 10, 11] and partly implemented in design tools (see references in [12]). A disadvantage of these methodologies and tools is that most of them are restricted to special architecture types (e.g. array processors consisting of one type of processing element (PE) connected by regular data dependences) or to a special class of algorithms (e.g. regular algorithms representable by nested loop programs). Furthermore, they do not support the complete design flow starting with the specification of the algorithms and ending with a netlist description at the gate level.

Due to these reasons the CAD-tool DECOMP has been developed to support the mapping of algorithms onto array processor architectures [13, 14]. The DECOMP requires PASCAL-descriptions [15] of the algorithms as input and produces EDIF-netlists [16] at register-transfer-level as output. Later developments lead to a new implementation of the frontend in the DECOMP which now is able to compile data independent algorithms [17] into dependence graphs (DGs) [12, 18]. The resulting DGs consist of different node types connected by nonregular data dependences. Thus, they cannot be mapped onto array architectures by the known design methodologies. To allow the mapping of these DGs a procedure for combining nodes of a different type into one PE has been derived [19], and in addition the mapping procedure proposed in [2] has been extended to handle nonregular data dependences. Currently the new mapping is implemented in the DECOMP.

The design process captured by the DECOMP cannot be performed in one step. Because of its complexity it has to be split into manageable design tasks each of them performing a specific design step. This results in a method referred to as *stepwise transformation*. A similar technique is known from high-level synthesis where it is applied to transform a behavioural description step by step into hardware (cf [20]). The purpose of this paper is to outline the formal approach underlying the stepwise transformation and its implementation. Furthermore, two data representations, one assigned to the algorithm level and the other assigned to the architecture level, are defined, based on which one of the main design steps of the transformation is explained in more detail. With the presented approach data independent algorithms can be mapped onto highly parallel array processor architectures. The main advantages of the presented transformation are its ability to process nonregular algorithms and its degree of automation.

In Section 2 of this paper the stepwise transformation is outlined, and in section 3 the data representations are introduced. One of the main design steps is explained in more detail in section 4. The implementation in the CAD-tool DECOMP is described in Section 5, and finally a design example is given in Section 6.

2. THE STEPWISE TRANSFORMATION

The design process of mapping a given algorithm onto an array processor architecture is performed in four phases. These are

1. a specification phase,
2. a compilation phase,
3. a mapping phase, and
4. an optimization phase.

The phases themselves are divided into smaller design tasks each of them performing a correctness preserving transformation. This means, without changing the I/O-behaviour of the algorithm. Thus, a given algorithm is step by step transformed into an array processor architecture. The phases and its design tasks are depicted in Fig. 1.

The *specification phase* consists of only one step which is the *Program development*. In this step the given algorithm is manually specified in a high-level language which is executable using standard compilers. Besides the algorithm the specification may contain an interface description specifying how the input data is provided and how the output data is required. In addition design constraints like maximum chip area and maximum delay times can be specified for the array processor or its PEs.

In the four steps of the *compilation phase* the description of the algorithm is modifed in a way that a dependence graph can be built from it. First, by application of compiler techniques [21] the given specification is symbolically executed [17] and the performed assignments are listed in the so-called *run time protocol* (RTP) [12]. For example the symbolic execution of the statements

```
a[1] := 0;
FOR i := 1 TO 2 DO
    b[2*i] := c[0];
    a[1] := a[1] + b[2 - i];        (1)
```

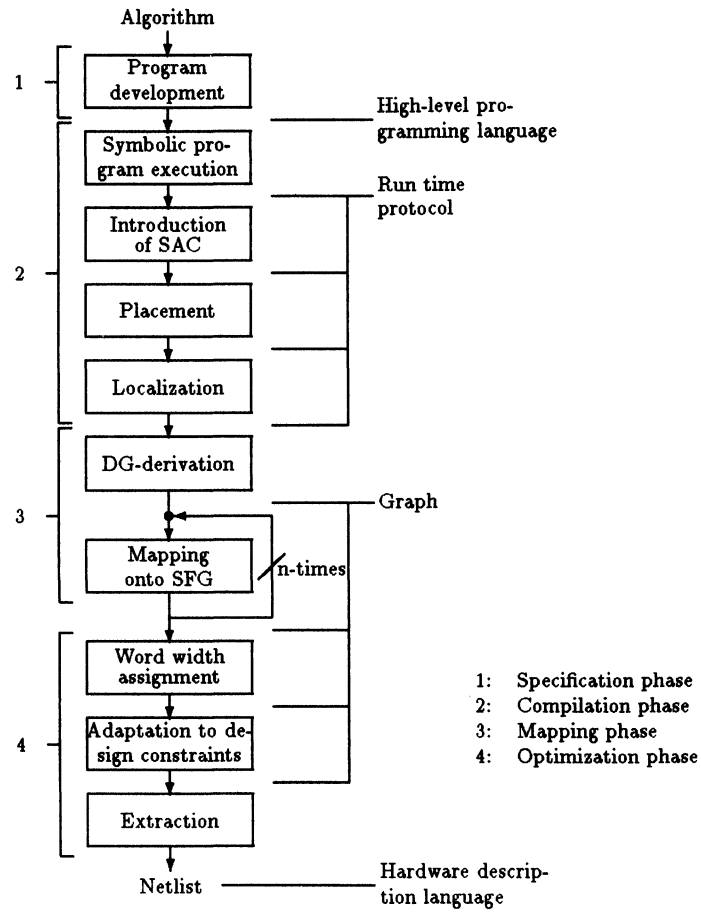


FIGURE 1 Transformation steps and representation of design data

leads to the following RTP:

$$
\begin{aligned}
a(1) &:= 0 \\
b(2) &:= c(0) \\
a(1) &:= (+a(1)b(1)) \\
b(4) &:= c(0) \\
a(1) &:= (+a(1)b(0))
\end{aligned} \tag{2}
$$

A unique extraction of the data dependences needed to build a DG only is possible if the RTP is given in single assignment code (SAC). In this code every variable is assigned one value only during the execution of the algorithm. Thus, in the second compilation step SAC is automatically introduced by adding an additional index where necessary. The RTP changes to

$$
\begin{aligned}
a(1\,0) &:= 0 \\
b(2) &:= c(0) \\
a(1\,1) &:= (+a(1\,0)b(1)) \\
b(4) &:= c(0) \\
a(1\,2) &:= (+a(1\,1)b(0))
\end{aligned} \tag{3}
$$

The allocation of the assignments to nodes in the DG is performed according to the indices of the variables on the left side of the assignments. Consequently, all left side variables must have the same number of indices. Furthermore, more regularity in the DG can be achieved if the nodes for the assignments are allocated under consideration of loop-counters in the input program or the similarity of the right sides of the assignments. Thus, in the third compilation step different *placement* algorithms can be applied to the RTP in order to achieve the highest possible degree of regularity and parallelism.

At last a *localisation* can be performed to avoid global data dependences in the resulting DG. During the localisation additional assignments which propagate the variables via neighbouring nodes only are introduced in the RTP. In the given example $c(0)$ is propagated via $c(2\,0)$ and $c(3\,0)$ to $b(4\,0)$. Finally the RTP may be as follows:

$$\begin{aligned} a(1\,0) &:= 0 \\ b(2\,0) &:= c(0) \\ c(2\,0) &:= c(0) \\ c(3\,0) &:= c(2\,0) \\ a(1\,1) &:= (+a(1\,0)b(1)) \\ b(4\,0) &:= c(3\,0) \\ a(1\,2) &:= (+a(1\,1)b(0)) \end{aligned} \tag{4}$$

The *mapping phase* consists of two steps which are the *DG-derivation* and the *mapping onto signal flow graphs* (SFGs). In the DG-derivation the RTP is transformed into a DG consisting of nodes and arcs (see example in Sec. 6). Then a mapping which is based on the multi projection method proposed in [2] is applied, and the DG is mapped onto a SFG thereby reducing the number of dimension by one. The mapping step can be applied to the resulting SFGs again to reduce the number of dimensions to a maximum of two. For a realization the advantage is given that the architectures can be implemented using only short and local connections between the PEs. Global connections on the chip should be avoided because their delays may dominate the delays of the gates.

In contrast to most of the known methodologies the proposed mapping has the advantage not to be restricted to homogeneous DGs with regular data dependences. It is also capable of mapping different nodes into the same PE by merging their internal structure as described in [19]. In addition, the method proposed in [2] is extended to handle DGs with irregular data dependences [22]. Furthermore, in the mapping step different projections can be applied. They result in a number of possible SFGs which differ in the number of PEs, the connections between the PEs, the data-I/O, the time needed for a complete computation of the algorithm, and other criteria. A suitable SFG fulfilling given design constraints is then selected and passed to the next phase.

In the *optimization phase* the selected SFGs are modified to array processor architectures. Because the SFGs are represented at the word level, first of all word widths for all arcs of the SFGs must be introduced. For this, the minimum and maximum values of the input signals are specified as a number range at the inputs of the SFG. Then a simulation function runs over the SFG calculating the minimum and maximum value for every arc in the graph as well as for every internal connection of the PEs.

Thereafter the derived architecture can be adapted to given design constraints. Not in every case for example it is possible to derive an architecture which requires the input data in the same way as it is provided by the external input interface. Therefore, register-multiplexer circuits for sorting the data coming from the input interface can be synthesized and put in front of the derived array processor. The problem of data supply for array processors has been studied in [23, 24].

The last step, the *extraction* of a netlist in a hardware description language, is performed by a direct conversion of the used data structure. The netlist is given at the register transfer level. The smallest blocks at this level are registers and arithmetic building blocks like adders and multipliers which can be generated using building block generators [25, 26] or other synthesis tools.

For implementation purpose it would be convenient to have the same data structures between every design step. Thus, the order of the steps can be changed or single steps can be left out. On the other hand the requirements from the algorithmic side of the transformation are totally different from those of the architectural side. Consequently, as shown in Fig. 1 the stepwise transformation is based on two different data representations, the RTP and a graph representation. The RTP is only used in the compilation phase. It is set-oriented, recursively defined (see Sec. 3.1) because it describes a sequence of assignments. In contrast the graph representation is blockoriented, hierarchically defined (see Sec. 3.2) because it describes the interconnections of hierarchically organized blocks.

3. REPRESENTATION OF DESIGN DATA

3.1 Algorithm Based Data Structure

The *run time protocol* which is produced by a symbolic program execution describes a given algorithm in a maximum expanded form. It holds one entry for every performed assignment and lists them in the

TABLE 1
Sets and elements used in the run time protocol

n	denotes the n-th processing step
$iv, \mathcal{IV}$	external input variable, set of all external input variables
$c, \mathcal{C}$	constant, set of all constants
$kv, \mathcal{KV}_n$	known variable, set of all known variables in the n-th processing step
$uv, \mathcal{UV}_n$	used variable, set of all used variables in the n-th processing step
pv_n	variable produced (assigned) in the n-th processing step

given execution order. Furthermore to all successive entries a discrete time step is assigned.

Formally a RTP can be described by grouping together the variables of the algorithm into different sets. Besides the set of external input variables ($\mathcal{IV}$) and the set of constants ($\mathcal{C}$) two more sets are defined at every discrete time step. One set ($\mathcal{KV}_n$) contains all variables which are known at a specific time step, the other set ($\mathcal{UV}_n$) contains all variables which are used by the assignment performed at that time step. For the definition of the RTP the sets and elements are named as given in Table 1. The RTP itself is defined in Def. D.1.

Def.: *Structure of the run time protocol* (D.1)

$$\mathcal{KV}_0 := \{kv | kv \in \mathcal{IV} \vee kv \in \mathcal{C}\}$$

$$pv_n := f_n(\mathcal{UV}_n)|_{\mathcal{UV}_n \subset \mathcal{KV}_n} \qquad \forall n \geq 0$$

$$\mathcal{KV}_{n+1} := \mathcal{KV}_n \cup \{pv_n\} \qquad \forall n \geq 0$$

This definition expands to the run time protocol shown in Table 2. Based on the external input variables iv and the used constants c the RTP is initialized at time step $n = 0$ with the set $\mathcal{KV}_0$. All input variables and constants can be collected in $\mathcal{KV}_0$ because in ordinary programs they are known at the beginning. Then, starting at $n = 0$ for every time step n the produced variables pv_n are calculated using the set $\mathcal{UV}_n$ of used variables which is a subset of $\mathcal{KV}_n$ containing all variables known at that time step. At the same time, the sets $\mathcal{KV}_{n+1}$ are calculated new for the following time steps. For that the variable pv_n and the set $\mathcal{KV}_n$ from the actual time step are used. Because in every time step the calculations use values from the directly preceding time step only, the RTP is said to be defined recursively.

TABLE 2
Maximum expanded run time protocol

Time step	Assignment
0	$pv_0 := f_0(\mathcal{UV}_0)\|_{\mathcal{UV} \subset \mathcal{KV}}$
·	· := .
·	· := .
n	$pv_n := f_n(\mathcal{UV}_n)\|_{\mathcal{UV}_n \subset \mathcal{KV}_n}$
·	· := .
·	· := .
$N-1$	$pv_{N-1} := f_{N-1}(\mathcal{UV}_{N-1})\|_{\mathcal{UV}_{N-1} \subset \mathcal{KV}_{N-1}}$
N	$pv_N := f_N(\mathcal{UV}_N)\|_{\mathcal{UV}_N \subset \mathcal{KV}_N}$

3.2 Architecture Based Data Structure

At the register transfer level array processors are represented by DGs and SFGs. These graphs consist of interconnected PEs which themselves consist of building blocks like adders and multipliers representing the basic operations of the implemented algorithm. The interconnections in DGs are arcs with zero delay, and in addition the DG is free of loops and cycles by definition. Interconnections of SFGs may contain registers (delays). Thus, SFGs may have loops and cycles with at least one register on them.

A design hierarchy in which DGs, SFGs, PEs, and building blocks can be represented as blocks is given by the following 3 levels:

1. **Graph-level** Representation of DGs and SFGs by PEs and their interconnections (also referred to as arcs).
2. **Processor-level** Representation of PEs by building blocks and their interconnections.
3. **Operator-level** Representation of building blocks. These are the smallest blocks at the register transfer level. Thus, their internal structure is not considered here.

DGs and SFGs are represented on the same level. Thus, in the following the statements for DGs are valid for SFGs as well except otherwise mentioned. Furthermore, nodes of DGs and PEs of SFGs are always referred to as PEs.

The blocks on each level formally are defined as 4-tuple as follows:

Def.: *Block description* (D.2)

$$bb = (A, \mathcal{B}, \mathcal{S}, \mathcal{F})$$

with

A as a 2-tuple describing the attributes of the block,

$\mathscr{B}$ as the set of subblocks used in the block.

$\mathscr{S}$ as the set of signals which represent the interconnections of the subblocks, as well as the inputs and outputs of the block,

$\mathscr{F}$ as the set of functions which represent the functionality of the block. The functions also represent the netlist of the block.

This definition was originally developed in [19] slightly modified as a model for complex processing elements. The attributes A in bb are defined as follows:

Def.: *Attribute* (D.3)

$$A = (typ, \mathscr{TSA})$$

with

typ as the identifier for the type of the block. It is one of the symbols DG, SFG, PE, or BB (building block),

$\mathscr{TSA}$ as a set of type specific attributes *tsa*

Each type specific attribute *tsa* is a 2-tuple of the following form:

Def.: *Type specific attribute* (D.4)

$$TSA = (att, val)$$

with

att indicating the attribute type. *att* also is used as keyword to distinguish different attributes.

val as the value of the attribute.

Type specific attributes for example are TSA_{DG} = (dim(2 4)) describing the index space of a DG or TSA_{PE} = (ind(11)) as the index point at which a PE is located in a DG. Different types of blocks may also have the same type of attribute. For example a building block may have the attribute TSA_{BB} = (area 42), and a PE which contains three of these building blocks may have the attribute TSA_{PE} = (area 126). For describing the reference to the attributes the following two functions are used:

$$f_{att}(TSA) = att$$
$$f_{val}(TSA) = val \tag{5}$$

As example: $f_{att}(TSA_{PE}) = f_{att}(\text{ind}(11))) = \text{ind}$

The set $\mathscr{F}$ in Def. D.2 contains functions F which are defined as follows:

Def.: *Function* (D.5)

$$F = (bb, \mathscr{S}^{in}, \mathscr{S}^{out})$$

with

$bb \in \mathscr{B}$ as a block which represents an operation like addition or more complex the operation of a PE (the blocks in a PE are called operators, too),

$\mathscr{S}^{in} \subset \mathscr{S}$ as the ordered set of input signals for the block,

$\mathscr{S}^{out} \subset \mathscr{S}$ as the ordered set of output signals for the block.

The given definition implicitly assumes that every block has at least one input and one output. Furthermore, the subblocks $bb \in \mathscr{B}$ and signals $s \in \mathscr{S}$ must have *unique names* inside a block because the netlist is given by referencing to these names. The order of the signals in the sets $\mathscr{S}^{in}$ and $\mathscr{S}^{out}$ is not important except when the functions are used for a symbolic verification at the operator level. Then the order is important for operators (blocks) which are not commutative (e.g., subtraction or division).

In addition, the signals $s \in \mathscr{S}$ optionally can be defined as a 3-tuple as follows:

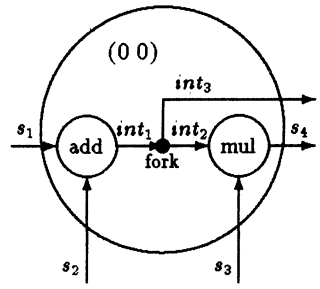
Def.: *Signal* (D.6)

$$s = (s_i, bb^{in}, bb^{out})$$

with bb^{in} as the block at which the signal starts and bb^{OUT} as the block at which it ends. Then the description of a block contains redundant information which for example can be used for a consistency check of the netlist.

A problem arises if internal signals of a block are allowed to be an output, too. In this case the output cannot be recognized automatically. In the presented model this is handled by *fork*-elements which have one input and more than one output. The function for a fork-element for example is $f_{fork} = (fork_1, \{s_1\}, \{s_2 s_3\})$.

As an example for the presented data representation the PE shown in Fig. 2 is given. This PE could be used in a DG as block bb_{0-0}.



$$PE_{o\_o} = (\ (PE, \{(ind, (0\ 0))\})$$
$$\{add\ mul\ fork\}$$
$$\{s_1\ s_2\ s_3\ s_4\ int_1\ int_2\ int_3\}$$
$$\{(add, \{s_1\ s_2\}, \{int_1\})$$
$$(mul, \{s_3\ int_2\}, \{s_4\})$$
$$(fork, \{int_1\}, \{int_2\ int_3\})\}\ )$$

$$bb_{0\_0} = (PE_{0\_0}, \{s_1\ s_2\ s_3\}, \{s_4\ int_3\})$$

FIGURE 2 Processing element and its formal representation

4. DESCRIPTION OF A MAIN TRANSFORMATION STEP

Due to space limitations a detailed explanation of the complete stepwise transformation does not fit into this paper. For this reason, the main design step *dg-derivation* has been selected for explanation because it is the interface between the two data representations. It transforms the run time protocol into a graph based, object-oriented data structure which can be easily converted into a netlist description.

Normally graphs are described by a 2-tuple $G = (\mathcal{N}, \mathcal{A})$ consisting of a set of nodes and a set of arcs. The arcs are described by $a = i \xrightarrow{a} j$ in which i and j are index points out of $\mathcal{N}$ (cf[2]). For the mapping of algorithms onto VLSI architectures in addition the external interfaces must be described. Due to this, the DGs derived during the stepwise transformation are described by a 4-tuple as given in Def. D.7.

Def.: *Dependence graph*
(and also signal flow graph) (D.7)

$$G := (\mathcal{N}, \mathcal{DEP}, \mathcal{DEP}^{in}, \mathcal{DEP}^{out})$$

with

- $\mathcal{N}$ as the set of nodes represented by their indices,
- $\mathcal{DEP}$ as the set of dependences connecting the nodes, usually referred to as arcs,
- $\mathcal{DEP}^{in}$ as the set of external input dependences,
- $\mathcal{DEP}^{out}$ as the set of external output dependences,

The dependences are given by a 4-tuple as follows:

Def.: *Dependences* (D.8)

$$dep := (start, end, name, delay)$$

Herein *start* and *end* denote the index points of the start and end node of the dependence. *name* is the individual name of the dependence and *delay* is the number of delays associated with the dependence. It should be noticed that the arcs in a DG always have zero delay by definition. The delay is specified because the definitions apply to SFGs as well. Furthermore, the values of such dependences are accessed by the following four functions:

Def.: *Access to dependences* (D.9)

$$f_s(dep) = start$$
$$f_e(dep) = end$$
$$f_n(dep) = name$$
$$f_d(dep) = delay$$

For the derivation of a DG first the set $\mathcal{PV}$ of produced variables and the set $\mathcal{UV}$ of used variables are built from a maximum expanded RTP (see Table 2) as follows:

$$\mathcal{PV} := \bigcup_{i=1}^{N} \{pv_i\} \tag{6}$$

$$\mathcal{UV} := \bigcup_{i=1}^{N} \mathcal{UV}_i \tag{7}$$

With these sets the external input and output arcs of the DG are given by Eq. 8 and 9, respectively. The symbol ε means that there is no value at this place.

As defined, the functions f_{ind} and f_{nam} return the index and the name of a variable.

$$\mathscr{DEP}^{in} := \{dep_{in} | dep_{in} = (\varepsilon, f_{ind}(pv_i), f_{nam}(uv), 0) \text{ with } pv_i := f_i(\mathscr{UV}_i) \wedge uv \in (\mathscr{UV}_i \cap (\mathscr{UV} \setminus \mathscr{PV}))\} \quad (8)$$

$$\mathscr{DEP}^{out} := \{dep_{out} | dep_{out} = (f_{ind}(pv_i), \varepsilon, f_{nam}(pv_i), 0) \text{ with } pv_i := f_i(\mathscr{UV}_i) \wedge pv_i \in (\mathscr{PV} \setminus \mathscr{UV})\} \quad (9)$$

The dependences between the nodes are determined by Eq. 10. Therein $i \succ k$ means the assignment i is performed before the assignment k.

$$\mathscr{DEP} := \{dep_{pv \to uv} | dep_{pv \to uv} = (f_{ind}(pv_i), f_{ind}(pv_k), f_{nam}(pv_i), 0) \text{ with } \mathrm{ipv_i} := \mathrm{f_i}(\mathscr{UV}_\mathrm{i}) \wedge \mathrm{pv_k} := \mathrm{f_k}(\mathscr{UV}_\mathrm{k}) \wedge \mathrm{i} \succ \mathrm{k} \quad \mathrm{pv_i} \in \mathscr{UV}_\mathrm{k}\} \quad (10)$$

The orientation of a dependence in the n-dimensional index space of the derived DG is given by the vector $\mathbf{r}$ as follows:

$$\mathbf{r} := f_e(dep) - f_s(dep) \quad (11)$$

Due to the placement according to the indices of the produced variables dependences with $\mathbf{r} = \mathbf{N}$ ($\mathbf{N}$ is the zero vector) may occur for the moment. Because such dependences are not allowed in a DG they later on are transformed into node internal signals.

After determination of the dependences the internal structure of the nodes is derived. Because this is a complicated operation first the index set $\mathscr{I}$ is developed with Eq. 12. It contains all index points of the n-dimensional index space at which a node of the DG is located.

$$\mathscr{I} := \{ind | \exists pv \in \mathscr{PV} \text{ mit } ind = f_{ind}(pv)\} \quad (12)$$

Then for every $ind \in \mathscr{I}$, i.e. for every node, a set $\mathscr{A}_{ind}$ is developed by application of Eq. 13. This set contains all assignments of the given run time protocol which produce a variable with the index ind. Thus, all these assignments are placed in the same node.

$$\mathscr{A}_{ind} := \{pv_i := f_i(\mathscr{UV}_i) | f_{ind}(pv_i) = ind\} \quad (13)$$

The sets $\mathscr{A}_{ind}$ contain assignments of arbitrary complexity described in prefix form, for example $(\ominus(\oplus uv_1[j_1 j_2] uv_2[k_1 k_2])\ uv_3[l_1 l_2])$ in which $\ominus$ and $\oplus$ denote any operators. For the translation of these assignments into functions of a block as defined in Def. 5 they first must be split into simple assignments only containing one operator which can be implemented by an arithmetic building block. This is performed by introducing additional variables which lateron lead to node internal signals. After the splitting the simple assignments can be directly translated into functions as shown in Eq. 14:

$$\begin{aligned} pv_i &:= f_i(\mathscr{UV}_i) \\ &\mapsto pv[i_1 i_2] := (\ominus(\oplus uv_1[j_1 j_2] uv_2[k_1 k_2]) uv_3[l_1 l_2]) \\ &\mapsto pv^*[i_1 i_2] := (\oplus uv_1[j_1 j_2] uv_2[k_1 k_2]) \\ &\quad \wedge pv[i_1 i_2] := (\ominus pv^*[i_1 i_2] uv_3[l_1 l_2]) \\ &\mapsto (\oplus, \{s_{uv} s_{uv}\}, \{s_{pv_i^*}\}) \\ &\quad \wedge (\ominus, \{s_{pv_i^*} s_{uv}\}, \{s_{pv_i}\}) \\ &\mapsto F_\oplus \wedge F_\ominus \end{aligned} \quad (14)$$

As the result of these transformations all sets $\mathscr{A}_{ind}$ are mapped into sets $\mathscr{A}^*_{ind}$ which describe the functionality of the nodes in the architecture based data structure (see Eq. 15)

$$\mathscr{A}^*_{ind} := \{F | F = (bb \mathscr{S}^{in} \mathscr{S}^{out}) \text{ with } \exists a \in \mathscr{A}_{ind} \text{ with } a \mapsto F_i \wedge F \wedge F_k \wedge \cdots\} \quad (15)$$

From the sets $\mathscr{A}^*_{ind}$ and with the functions

$$\begin{aligned} f_{bb}(F) &= bb \\ f_{\mathscr{S}^{in}}(F) &= \mathscr{S}^{in} \\ f_{\mathscr{S}^{out}}(F) &= \mathscr{S}^{out} \end{aligned} \quad (16)$$

a set of node descriptions can be derived as follows:

$$\mathscr{N} := \{n | n = (A_n \mathscr{B}_n \mathscr{S}_n \mathscr{F}_n) \text{ with } A_n = (PE\{(\text{ind } ind)\}) \text{ with } ind \in \mathscr{I} \wedge \mathscr{B}_n = \bigcup_{F \in \mathscr{A}^*_{ind}} \{f_{bb}(F)\} \wedge \mathscr{S}_n = \bigcup_{F \in \mathscr{A}^*_{ind}} (f_{\mathscr{S}^{in}}(F) \cup f_{\mathscr{S}^{out}}(F)) \wedge \mathscr{F}_n = \mathscr{A}^*_{ind}\} \quad (17)$$

The derived sets completely describe the DG which becomes

$$G := \left(\mathcal{N}, \mathcal{DEP}, \mathcal{DEP}^{in}, \mathcal{DEP}^{out}\right) \quad (18)$$

Finally the description of the DG required for the hierarchically defined, architecture based data representation (see Def. 2) is derived by the following equation:

$$\begin{aligned} DG &:= (A_{DG}, \mathcal{B}_{DG}, \mathcal{S}_{DG}, \mathcal{F}_{DG}) \\ \text{with} \quad A_{DG} &= (DG, \{(\dim, dim)\}) \\ &\wedge \mathcal{B}_{DG} = \mathcal{N} \\ &\wedge \mathcal{S}_{DG} = \mathcal{DEP} \cup \mathcal{DEP}^{in} \cup \mathcal{DEP}^{out} \\ &\wedge \mathcal{F}_{DG}\left\{F | F = \left(pe, \mathcal{S}_{pe}^{in}, \mathcal{S}_{pe}^{out}\right)\right. \end{aligned} \quad (19)$$

$$\begin{aligned} \text{with} \quad pe \in \mathcal{N} \wedge pe &= (A_n, \mathcal{B}_n, \mathcal{S}_n, \mathcal{F}_n) \\ &\wedge \mathcal{S}_{pe}^{in} = \mathcal{IN}_{pe} \\ &\wedge \mathcal{S}_{pe}^{out} = \mathcal{OUT}_{pe}\} \end{aligned}$$

In Eq. 19 the two sets $\mathcal{IN}_{pe}$ and $\mathcal{OUT}_{pe}$ are derived from the functions of the PE as follows:

$$\begin{aligned} \mathcal{IN}_{pe} = \{s | \exists F \in \mathcal{F}_n \text{ with } s \in f_{\mathcal{S}^{in}}(F) \\ \wedge \nexists F \in \mathcal{F}_n \text{ with } s \in f_{\mathcal{S}^{out}}(F)\} \end{aligned} \quad (20)$$

$$\begin{aligned} \mathcal{OUT}_{pe} = \{s | \exists F \in \mathcal{F}_n \text{ with } s \in f_{\mathcal{S}^{out}}(F) \\ \wedge \nexists F \in \mathcal{F}_n \text{ with } s \in f_{\mathcal{S}^{in}}(F)\} \end{aligned} \quad (21)$$

Furthermore, the dimension *dim* of the DG is given by

$$dim := \max_{\forall i = 1, \ldots, N} (n(pv_i)) \quad (22)$$

where $n(pv_i)$ denotes the number of indices of the variable pv_i. In the stepwise transformation the dimension is already determined during the placement in which the assignments of the RTP are assigned to index points of a n-dimensional index space by consistently changing the indices of the produced variables.

A synthesis algorithm for the derivation of DGs from the RTP has to apply the given equations in the order as described above. The main advantage of this approach is that the basic sets $\mathcal{PV}$, $\mathcal{UV}$ and the respective $\mathcal{A}_{ind}$ are derived by linear processing of the run time protocol once only. The other sets are calculated from these basic sets without time consuming search operations. Furthermore, if regular algorithms like matrix-matrix-multiplication are processed the basic sets could be derived immediately during the symbolic execution in the compilation phase. This is very fast, but only possible if no special placement is required.

Another point is that the placement according to the indices of the produced variables may result in DGs with cycles which are not allowed by definition. But, these cycles only exist at the graph level. Due to the single assignment property, which is automatically introduced during the compilation phase they do not exist at the processor level. Therefore, they can be broken up easily by assigning the concerned operations to other nodes in the DG. Furthermore, in the presented approach the existence of such cycles is already checked during the placement at the compilation phase. Thus the derived DGs are free of loops and cycles.

5. THE CAD-TOOL DECOMP

To show the feasibility of the formal approach presented in this paper the stepwise transformation has been implemented in the CAD-tool DECOMP. COMMON LISP [27] has been used as implementation language because it offers various possibilities for the implementation of object oriented data structures as required by the presented formal approach. In addition, it is well suited for rapid prototyping. The program structure of the DECOMP is shown in Fig. 3.

The transformation of a given algorithm starts with an input description in the high-level language PASCAL. Besides the algorithm this input description also may contain a specification of the external interfaces and given design constraints. The first transformation step in the compilation phase, the symbolic execution, is performed by application of compiler techniques [21]. Precise, the frontend compiler of the DECOMP analyses the input description based on a grammar describing the input language. Thus, by changing this grammar, other input languages can be implemented easily without changing the source code of the compiler.

The symbolic execution as well as the introduction of single assignment code (SAC), the localisation, and the placement (see Fig. 1) together are implemented in the program component *compilation* which produces a localized and placed RTP in SAC

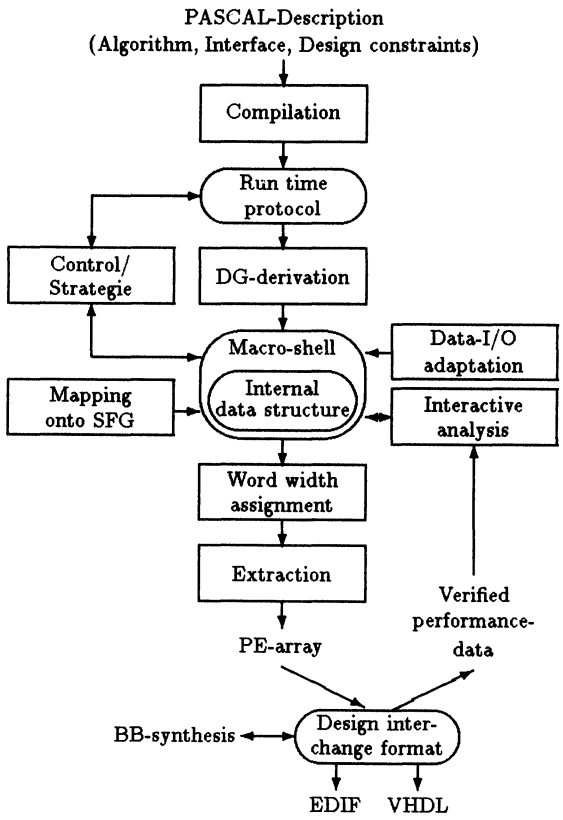


FIGURE 3 The structure of the DECOMP

as output. This RTP is input to the program component *DG-derivation* which translates it into the internal data structure. The internal data structure is able to represent graphs (DGs and SFGs) as well as PEs and their building blocks. Furthermore, to avoid confusion with the design data the internal data structure is only accessible via a *macro-shell*.

The second step of the mapping phase, the *Mapping onto SFG*, is implemented in an own program component providing various functions for projecting a given DG onto different SFGs. Additionally, it provides functions for comparison and verification of the SFGs. For example the verification function simulates the data flow in the SFGs by reading input data from a file (the same file as used by the original PASCAL program) into the SFG and then calculating the output values of each PE. The PEs are processed in the order given by the inherent schedule of the SFG as many times as necessary to calculate all output data. The output data then can be compared to the data produced by running the PASCAL program.

From the optimization phase the two steps 'word width assignment' and 'extraction' are implemented in two successive program components. The *word width assignment* takes a SFG from the internal data structure and after introducing the maximum needed word widths it passes the SFG to the *extraction*. The extraction generates a special *design interchange format* (DIF) which is used to communicate with other design tools.

Via the DIF a building block generator [25, 26] is connected to the DECOMP. Thereby, the synthesis of the required building blocks can be performed outside the DECOMP and the performance data of the generated building blocks can be written back into the DIF. Furthermore, different netlist converters are implemented allowing the translation of the DIF into the standards EDIF and VHDL Thus, further design steps like simulation and layout synthesis can be performed with commercially available CAD-tools.

The performance data of the generated building blocks can be transferred back into the internal data structure of the DECOMP. Then, based on this data the created architectures, or rather SFGs, can be interactively analysed with respect to the specified design constraints. The program component *interactive analysis* provides functions e.g., for the calculation of area and delay parameters of the architectures. If an architecture does not fulfill required constraints it can be modified and extracted again. This design cycle implements the optimization step 'adaptation to design constraints'.

In addition, the program component *data I/O-adaptation* provides functions for generation of register-multiplexer circuits which adapt the input interface of a designed architecture to a given external interface. The generated circuits are represented in the same data structure as the SFGs. Therefore, the same design steps can be applied to them.

The program components of the DECOMP allow a straightforward conversion of an algorithm into an architecture as intended by the formal approach of the stepwise transformation. The components of the DECOMP can be used interactively or in an automated way. In the latter case the program component *control/strategy* performs an experience based heuristic search to find architectures close to given design constraints. To allow this the control/strategy has access to all data structures.

6. A DESIGN EXAMPLE

In this section a design example for the stepwise transformation of an algorithm into an array processor architecture is given. Here the blockmatching

```
PROGRAM blockmatching;
CONST nn  = 3;
VAR   <-- variable declarations -->
BEGIN
 <-- specification of design constraints -->
 <-- description of the input interface -->
 x_n[1] := maxint;
 FOR n := 1 to nn DO
  BEGIN
   x_m[1] := maxint;
   FOR m := 1 to nn DO
    BEGIN
     x_i[1] := 0;
     FOR k := 1 to nn DO
      BEGIN
       x_k[1] := 0;
       FOR i := 1 to nn DO
        IF (i=nn) THEN
         IF (k=nn) AND (i=nn) THEN
          IF (k=nn) AND (i=nn) AND (m=nn) THEN
           BEGIN
            x_k[1] := x_k[1] + abs(x_in[i-1,k-1] - y_in[i+n-2,k+m-2]);
            x_i[1] := x_i[1] + x_k[1];
            x_m[1] := min(x_m[1],x_i[1]);
            x_n[1] := min(x_n[1],x_m[1]);
           END
          ELSE
           BEGIN
            x_k[1] := x_k[1] + abs(x_in[i-1,k-1] - y_in[i+n-2,k+m-2]);
            x_i[1] := x_i[1] + x_k[1];
            x_m[1] := min(x_m[1],x_i[1]);
           END
         ELSE
          BEGIN
           x_k[1] := x_k[1] + abs(x_in[i-1,k-1] - y_in[i+n-2,k+m-2]);
           x_i[1] := x_i[1] + x_k[1];
          END
        ELSE
          x_k[1] := x_k[1] + abs(x_in[i-1,k-1] - y_in[i+n-2,k+m-2]);
      END; END; END;
 <-- description of the output interface -->
END.
```

FIGURE 4 Algorithmic part of the input description

algorithm from the area of image processing has been chosen (cf [28]). It is shown in Eq. 23.

$$U = min_{m,n} \sum_{i=1}^{N} \sum_{k=1}^{N} |x_{i,k} - y_{i+m,k+n}| \qquad (23)$$

$$V = (m,n)|U \qquad n,m = -\frac{N}{2} \cdots \frac{N}{2}$$

n and m can only take integer values of the specified interval. In the following only the calculation of U is considered. First of all, the algorithm has to be formulated in the high-level language PASCAL which is required as input for the DECOMP. The algorithmic part of the input description which has been formulated for $N = 3$ is shown in Fig. 4. It consists of four nested loops, two of them for the calculation of the sums over i and k and two of them for calculating the sums over the variable displacements in m and n. The input description neither is localized nor is it given in single assignment code. It should be noticed that the declaration of all variables $x\_\ldots$ and $y\_\ldots$ as array of the length one is necessary due to the prototype implementation of the DECOMP-compiler which otherwise cannot distinguish between variables used for calculation of data and variables used as loop counter. Further, for verification purpose the input description can be executed using a standard PASCAL compiler.

The given input description is translated into a RTP by the DECOMP-compiler which at the same time automatically introduces SAC. Then a placement algorithm is applied which consistently changes

TABLE 3
Parts of the RTP for the blockmatching example

0	((X_K (1111))	(+0(ABS(_(X_IN(00))(Y_IN(00))))))
1	((X_K (2111))	(+(X_K(1111))(ABS(_(X_IN(10))(Y_IN(10))))))
2	((X_K (3111))	(+(X_K(2111))(ABS(_(X_IN(20))(Y_IN(20))))))
3	((X_I (3111))	(+0(X_K(3111))))
4	((X_K (1211))	(+0(ABS(_(X_IN(01))(Y_IN(01))))))
5	((X_K (2211))	(+(X_K(1211))(ABS(_(X_IN(11))(Y_IN(11))))))
6	((X_K (3211))	(+(X_K(2211))(ABS(_(X_IN(21))(Y_IN(21))))))
7	((X_I (3211))	(+(X_I(3111))(X_K(3211))))
8	((X_K (1311))	(+0(ABS(_(X_IN(02))(Y_IN(02))))))
9	((X_K (2311))	(+(X_K)(1311))(ABS(_(X_IN(12))(Y_IN(12))))))
10	((X_K (3311))	(+(X_K(2311)(ABS(_(X_IN(22))(Y_IN(22))))))
11	((X_I (3311))	(+(X_I(3211))(X_K(3311))))
12	(X_H (3311))	(HIN 999999 (X_I(3311))))
...	...	...
116	((X_K (3333))	(+(X_K(2333))(ABS(_(X_IN(22))(Y_IN(44))))))
117	((X_I(3333))	(+(X_I(3233))(X_K(3333))))
118	((X_H(3333))	(HIN (X_H(3323))(X_I(3333))))
119	((X_N(3333))	(HIN(X_N(3332))(X_H(3333))))

the indices of the produced variables in the RTP according to the loop counters. Thus, the regularity inherently given in the blockmatching algorithm is exploited. Parts of the resulting RTP are shown in Table 3.

With the DG-derivation as described in Sec. 4 a 4-dimensional DG is derived. It consists of 81 nodes of 4 different types. So, the assignments 116 to 119 are placed in the same node because the indices of their produced variables are the same. Due to the regularity of the blockmatching algorithm, the nodes of the DG are connected in a regular manner allowing several different projections. With the program component *mapping onto SFG* the multiprojection described in [2] is applied and the DG is projected three times. The used projection and schedule vectors are as follows:

1. projection: $\mathbf{d}_1 = (1000)$ $\mathbf{s}_1 = (1000)$
2. projection: $\mathbf{d}_2 = (100)$ $\mathbf{s}_2 = (100)$
3. projection: $\mathbf{d}_3 = (10)$ $\mathbf{s}_3 = (11)$

Every projection reduces the dimension of the graph by one. Thus, the resulting SFG which is shown in Fig. 5 has one dimension and consists of three different PEs. The numbers at the connections denote the number of delays (registers) associated with them. Besides the two inputs **x_in** and **y_in** for the image data every PE has several inputs for intermediate values calculated during processing an image. In addition, every PE has control inputs **s_...** which determine whether a signal (e.g., **x_i**) is taken from the feed-back loop or from an external input. The external inputs are used for the input of start values (e.g., 0 or **max int**) for the intermediate sums. The resulting minimum U for Eq. 23 is produced by the output **x_m** of the third PE after 43 clock periods.

The PE at the index (1) is shown in Fig. 6. It consists of two multiplexers and five arithmetic building blocks. The path from the inputs **x_in** and **y_in** via the blocks $-$, *ABS*, $+$, $+$, and *MIN* to the output represents the calculation of Eq. 23. The two multiplexers are used to select an external input value or one stored in the registers of the feed-back loops. The sequences of control signals for the mul-

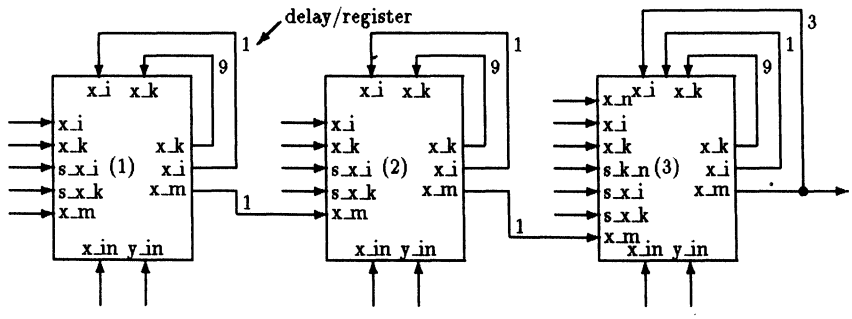


FIGURE 5 Architecture for the blockmatching example

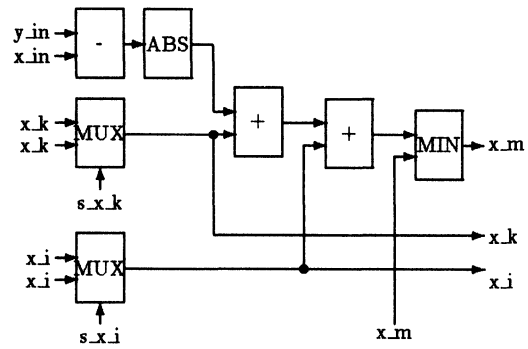


FIGURE 6 Internal structure of processing element at the index 1

tiplexers are automatically derived during the multiprojection. It is also possible to verify the SFG at the word-level. Therefore a verification function is implemented in the DECOMP which simulates the SFG using the same input data as the PASCAL description. The SFG represents an architecture for the given algorithm if the output data of the verification function is equal to the output data of the PASCAL program.

In the final design steps word widths for all connections are automatically introduced based on specifications given in the input description. Such specifications contain a number range or a concrete word width for the external input signals. After this the SFG is translated into an intermediate *design interchange* format from which netlists in EDIF or VHDL can be generated.

The example has been processed with a prototype implementation of the DECOMP on a VAXstation 4000/60. The compilation into a DG and the straightforward multiprojection onto the SFG together were performed in approximately 40 CPU seconds, the verification took approximately 17 CPU seconds, and the introduction of the maximum needed word widths approximately 27 CPU seconds.

7. CONCLUSIONS

The presented formal method performs an automatic mapping of data independent digital signal processing algorithms onto array processor architectures. It covers the complete design flow from algorithmic specifications in a high-level language to the generation of netlist descriptions at the register transfer level. The presented approach extends known design methodologies because it can handle inhomogen dependence graphs with nonregular data dependences. Thereby, also irregular algorithms can be mapped onto an architecture which is as regular as the given algorithm. As basic architecture type array processors have been chosen because they provide high computation power due to a massive application of pipelining and parallel processing. In addition, due to their regularity, they are well suited for an automatically performed design process. The presented *stepwise transformation* divides the complete design task into smaller parts which can be independently handled by a CAD-tool.

The feasibility of the approach has been shown by the implementation in the CAD-tool DECOMP. With the DECOMP it is possible to translate an executable PASCAL-description of a data independent algorithm into a dependence graph. Single assignment code is automatically introduced and global connections are localized. The dependence graphs can be mapped onto different signal flow graphs which are modified to architectures by the introduction of maximum needed word widths. The DECOMP supports comparison of the derived architectures to each other and to given design constraints. Thus, an optimal architecture can be selected and translated into an intermediate design interchange format from which netlists at the register transfer level can be generated in EDIF or VHDL. Furthermore, the interfaces of the synthesized architectures can be adapted to an external data supply by generation of register-multiplexer circuits. Supporting designers at the architecture level, which currently is not sufficiently supported by commercial design tools, the DECOMP is an important step towards design automation. In addition, the given blockmatching example shows that the derived architectures are comparable to those known from the literature.

With the DECOMP the designers are enabled to explore interactively different architectures in short time. Herein, the problem, which also is present in other design methodologies, is the existence of an exhaustive number of different design solutions. It is not possible to control an automatically performed mapping process in such a way that it maps the algorithm straightforward onto an optimal architecture. Therefore, the decisions in the design steps (e.g., which placement algorithm has to be chosen, how the localisation has to be performed, and which projection vectors have to be applied) are based on heuristic and empirical knowledge. For example, the projection directions are chosen parallel to the coordinate axes. Therefore, in the future, criteria must be developed which allow estimation of the performance data of the architectures (e.g., area, time,

number of inputs...) at a very early design stage. Currently a design strategy is developed which based on such criteria allows the further automation of the interactively performed design process in the DECOMP.

Acknowledgment

This work has been supported by the Fraunhofer Gesellschaft contract number T/RF33/K0011/K1311.

References

[1] P. Pirsch, "VLSI Architectures for Digital Video Signal Processing". P. Dewilde and S. Vandewalle, editors, *Computer Systems and Software Engineering (State of the Art)*, pp. 65–100. Kluwer Academic Publishers, Dordrecht, 1992.

[2] S.Y. Kung, *VLSI-Array Processors*, Prentice Hall Information and System Sciences Series, Englewood Cliffs, Thomas Kailath edition, 1988.

[3] J.A.B. Fortes, K.S. Fu, and B.W. Wah, "Systematic Approaches to the Design of Algorithmically Specified Systolic Arrays". *Proc. ICASSP*, pp. 300–303, March 1985.

[4] D.I. Moldovan, "On the Design of Algorithms for VLSI Systolic Arrays". *Proc. IEEE*, vol. 71, pp. 113–120, January 1983.

[5] P. Quinton, "Automatic Synthesis for Systolic Arrays from Uniform Recurrent Equations". *Proc. of 11th Annual Intern. Symp. on Computer Architecture*, pp. 208–214. IEEE, June 1984.

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

[7] L. Thiele, "On the Hierarchical Design of VLSI Processor Arrays". *Proc. of IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 3, pp. 2517–2520, 1988.

[8] J. Bu, *Systematic Design of Regular VLSI Processor Arrays*, PhD thesis, Delft University of Technology, May 1990.

[9] J. Bu, Ed F. Deprettere, and L. Thiele, "Systolic Array Implementation of Nested Loop Programs". E.F. Deprettere, editor, *Algorithms and Parallel VLSI Architectures*, pp. 111–118. Elsevier, Amsterdam, 1991.

[10] J.H. Moreno and T. Lang, "Matrix Computations on Systolic-Type Meshes (An Introduction to the Multimesh Graph Method)", *IEEE Computer*, pp. 32–51, April 1990.

[11] V. Van Dongen, *From Systolic to Periodic Array Design*, PhD thesis, Université Catholique de Louvain, Louvain, Belgium, January 1991.

[12] U. Vehlies and U. Seiler, "The Application of Compiler Techniques in Systolic Array Design". *Proc. ISCAS*, vol. 1, pp. 240–243. IEEE, June 1991.

[13] U. Vehlies, "DECOMP-A Program for Mapping DSP-Algorithms onto Systolic Arrays". G.M. Megson, editor, *Transformational Approaches to Systolic Design*. Chapman and Hall, London, 1994.

[14] U. Vehlies and A. Crimi, "A Compiler for Generating Dependence Graphs of DSP-Algorithms". E.F. Deprettere and A.J. Van der Veen, editors, *Algorithms and Parallel VLSI Architectures*, vol. B, Proceedings, pp. 319–328. Elsevier, Amsterdam, 1991.

[15] D. Cooper, *Standard Pascal-User Reference Manual*, W.W. Norton & Company, New York, 1983.

[16] EDIF Steering Committee, *EDIF: Electronic Design Interchange Format Version 200. Washington*, 1987.

[17] A. Berlin and D. Weise, "Compiling Scientific Code Using partial Evaluation", vol. 23(12), *Computer*, pp. 25–37, December 1990.

[18] U. Vehlies, "The Derivation of Dependence Graphs from PASCAL Programs for Array Processor Design". P. Quinton and Y. Robert, editors, *Algorithms and Parallel VLSI Architectures II*, pp. 371–376. Elsevier, Amsterdam, 1991.

[19] U. Vehlies, "Mapping Different Node Types of Dependence Graphs into the same Processing Element". M. Valero, S.-Y. Kung, T. Lang, and J.A.B. Fortes, editors, *Proc. Application Specific Array Processors 1991*, pp. 72–86. IEEE Computer Society Press, Los Alamitos, 1991.

[20] R. Camposano, "From Behaviour to Structure: High-level Synthesis", (10) *IEEE Design & Test of Computers*, pp. 8–19, October 1990.

[21] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers-Principles, Techniques and Tools*, Addison Wesley, 1986.

[22] U. Vehlies, "Mapping Dependence Graphs with Irregular Data Dependences onto signal flow graphs". Technical report, Laboratorium für Informationstechnologie, University of Hannover, 1993.

[23] M. Schönfeld, P. Pirsch, and M. Schwiegershausen, "Synthesis of Intermediate Memories Needed to Handle the Data Supply of Processor Arrays". W. Rosenstiel, editor, *Fifth International ACM & IEEE Workshop on High-Level Synthesis*, pp. 21–28. March 1991.

[24] M. Schönfeld, P. Pirsch, and M. Schwiegershausen, "Synthesis of Intermediate Memories for the Data Supply to Array Processors". P. Quinton and Y. Robert, editors, *Algorithms and Parallel VLSI Architectures II*, pp. 365–370. Elsevier, Amsterdam, 1991.

[25] A. Münzner and P. Pirsch, "BADGE-Building Block Adviser and Generator". *Proc. ISCAS*, vol. 3, pp. 1887–1890, May 1989.

[26] A. Münzner, "Building Block Generation considering the inherent Hierarchy of Arithmetic Operations". *Proc. of the IFIP Working Conference on Logic and Architecture Synthesis*, pp. 277–286, May 1990.

[27] G.L. Steele Jr., *COMMON LISP*, Digital Press, 1984.

[28] T. Komarek and P. Pirsch, "Architectures for Block Matching Algorithms", vol. 36(10), *IEEE Transactions on Circuits and Systems*, pp. 1301–1308, October 1989.

Biography

DR. UWE VEHLIES was born July 7, 1960 in Hameln, Germany. After taking his school-leaving exam in 1980 he completed his military service. From 1981 to 1988 he studied electrical engineering at the University of Hannover concentrating on communications processing. He received the diploma degree in July 1988. From August 1988 to June 1993 he was research assistant at the Information Technology Laboratory in Hannover. Working in the group of Prof. P. Pirsch he was involved in the development of CAD-tools especially for the design of array processor architectures. In August 1993 he got his PhD. Since then he is involved in the development and advice of software for business applications.