Jacobi-specific processor arrays

Hylke W. van Dijk *, *Gerben J. Hekstra* † *and Ed F. Deprettere* †

*	†
Philips Semiconductors (PCALE)	Dept. of Electrical Engineering
Building BE3	Delft University of Technology
P.O.Box 218 5600MD Eindhoven	2628 CD Delft

ABSTRACT

We present a processor and a compiler for prototyping array implementations of algorithms from the class of Jacobi algorithms. We use adaptive matrix QR decomposition as illustrative example.

1. INTRODUCTION

In VLSI signal processing, there is a tendency toward the design of family specific processors. The (algorithm) family may consist of a number of derivatives of a single 'mother' algorithm such as the family of code excited linear predictive speech coders [1]. For this and similar classes, one can envisage a processor which is optimal for the family as a whole. The family may also be broader. Thus the video signal processor (VSP) - or rather an array of such processors - is suited for the implementation of a whole set of video algorithms. Architectures of family specific processors may be of VLIW type, may be two-level multiprocessor architectures or even native microprocessors. The family we consider in this paper is a set of algorithms known as Jacobi algorithms. Examples of Jacobi algorithms are (time-update) QR decomposition, singular value decomposition, and various types of subspace tracking algorithms [2] [3] and adaptive orthogonal filters [4] [5]. One reason for chosing this family is that it provides a nice example of regular/irregular co-design. Most flow graph simulation environments and behavioral design systems tend to be generic in that they do not distinguish between regular and irregular graphs. Regular flow graphs have special properties which can be exploited to make their specification, description and manipulation elegant and tractable. Design systems that focus on such flow graphs do exist but they in turn tend to exclude irregularity. This paper is an attempt to bring the two domains

1063-6862/96 $5.00 © 1996 IEEE

together. The idea The idea of why and how designing a specific processor for this family of algorithms was presented in [6] where it was dubbed the *Jacobi processor*. This processor is actually a co-processor of a two-level multiprocessor architecture - the *Jacobi array* which consists of a number of communicating such co-processors and a supervising control generator. The program for the generator is compiled by letting a design system behave as compiler. Thus instead of designing an architecture for the parallel implementation of a given algorithm, the system outputs a control program by taking in an algorithm from the family of algorithms and an implementation structure from the set of feasible two-level multiprocessor architectures.

Most software programs implementing (time-update) Jacobi algorithms are nested loop programs (NLP) which are of the following form

for $k = 1 \ : \ 1 \ : \ K$

$$Y_k \leftarrow \left(\prod_{s=0}^{N_s-1} A_{(i_s,j_s)}(\alpha_k,\beta_k,\theta_k)_{(i_s,j_s)}\right) X_k \left(\prod_{t=0}^{N_t-1} B_{(i_t,j_t)}(\gamma_k,\delta_k,\phi_k)_{(i_s,j_s)}\right)^T$$

end

Where

(1) $X_k = \begin{bmatrix} Y_{k-1} \\ x_k^T \end{bmatrix}$

with x_k^T a row vector and $Y_k \in \mathbb{R}^{(m-1)\times n}$.

(2) $A_{(i,j)}(\alpha_k,\beta_k,\theta_k)_{(i_s,j_s)}$ and $B_{(i,j)}(\gamma_k,\delta_k,\phi_k)_{(i_s,j_s)}$ are identity-embedded 2×2 matrices $A(\alpha,\beta,\theta)$ and $B(\gamma,\delta,\phi)$, respectively, which are both of the general form

$$M(\alpha,\beta,\varphi) = M(\varphi) \begin{bmatrix} e^{j\alpha} & 0 \\ 0 & e^{j\beta} \end{bmatrix}$$

with $M(\varphi)$ any member of a set of *elementary coordinate transformation* (ECT) { $J(\alpha)$, $G(\alpha)$, $H(\alpha)$ } defined by

$$J(\varphi) = \begin{bmatrix} cos\varphi & -sin\varphi \\ sin\varphi & cos\varphi \end{bmatrix}, \ G(\varphi) = \begin{bmatrix} 1 & 0 \\ -\varphi & 1 \end{bmatrix}, \ H(\varphi) = \begin{bmatrix} cosh\varphi & sinh\varphi \\ sinh\varphi & cosh\varphi \end{bmatrix}$$

Of course, in any specific instance of the algorithm, the set of indices i_s, i_t, j_s and j_s, the boundary values N_s and N_t as well as the rules to determine the parameters α_k, β_k, θ_k, γ, δ_k and ϕ_k of the matrices $A_{(i_s,j_s)}$ and $B_{(i_t,j_t)}$ have to be specified. We say that an ECT is conditional if the angles have to be determined from the matrix X, it is unconditional otherwise. The processor - called Jacobi processor in [6] - has a hierarchical or layered structure with generic layer model as shown in figure 1.

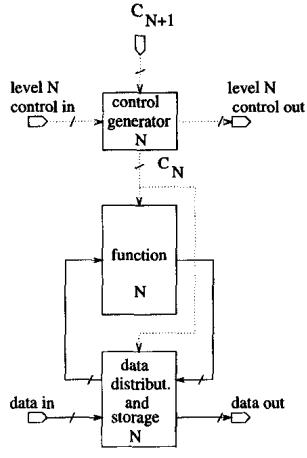

Figure 1. Hierarchical or layered control model.

The bottom most layer in the hierarchy bears the *core function*. One layer up is found the *kernel functions*, the ECTs, which are composed by means of the core function. The layer above the ECT-level is the *compound function* layer. On this layer are mapped irregular flow graphs whose nodes are kernel functions. One level up - the highest layer of the Jacobi co-processor - is the layer of *cluster functions*. These are tiles from a regular dependence graph whose node functions are compound functions. A network of such clusters, that is, a flow graph whose nodes are tiles is implemented in a yet higher layer which consists of communicating Jacobi co-processors. This constitutes the Jacobi processor. An array of Jacobi processors can form a next layer. The highest layer is the system level. Figure 2 summarizes the distinctive hierarchical levels of the architecture. The functionality of the layers is given through sets of functions $F^{(i)}$, one for each level. Level $i = 0$ lays at the transition between regular and irregular flow graphs. Thus, $F^{(0)}$ is the set of all feasible (compound) functions[1] that can appear in the dependence graph of a Jacobi algorithm. Similarly, $F^{(-1)}$ is the set of all possible kernel functions that appear in the refinement of the compound functions and that are known

[1]By functions, we mean *applicative state transition functions* (ASTs) as defined in [7] or [8].

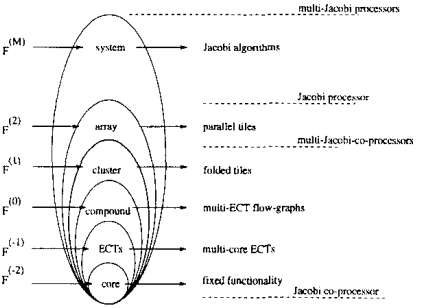

Figure 2. The hierarchical implemetation structure for execution of Jacobi algorithms.

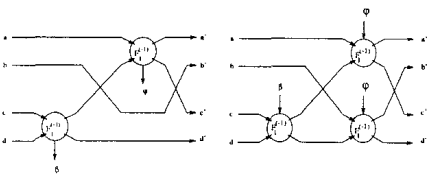

Figure 3. Conditional (left) and unconditional complex rotations

to the compiler. This set is the set $\{J(\varphi),\ G(\varphi),\ H(\varphi)\}$ - implemented in Cordic arithmetic[9] [10], and the set of micro-rotations defined in [11] and [12].

Control is expressed in terms of control vectors $\alpha^{(i)}$. This vector selects the appropriate function $f^{(i)}$ from the set $F^{(i)}$. Take for example level (0). This is the dependence graph level. Jacobi algorithm dependence graphs are generally piecewise regular graphs, each piece or domain being characterized by a linearly bounded lattice [13]. Within a domain, all nodes have identical functionality. Therefore, there is one vector $\alpha_j^{(0)}$ for each domain D_j. The unique function on this domain is $f_j^{(0)} = F^{(0)}(\alpha_j^{(0)})$. See [6] for more details. Examples of elements from the set $F^{(0)}$ are shown in figure 3. They are both complex rotations $M(\alpha, \beta, \varphi)$, expressed in terms of real rotations $J(\beta)$ and $J(\varphi)$. The real rotations themselves belong to the set $F^{(-1)}$ and are built on shift-and-add operations from $F^{(-2)}$.

Levels (0), (−1) and (−2) are jointly visualized in figure 4 for the adaptive QR example [14].

The level (0) graph is a portion of a dependence graph wherein the dark nodes are conditional Givens rotations and the gray ones are unconditional Givens rotations. Layer (−1) provides the two complex rotations, see also figure 3. A possible unfolded flow graph of the two real rotations on level (−2) are also shown.

Similarly, levels (0), (1) and (2), for the same example, are jointly shown in figure 5.

In this figure, the dependence graph (level (0)), is partitioned by tiling the graph with parallelepipeds. A single tile resides on level (1) and is folded into a cluster of the cluster layer. Another tiling, one hierarchical level up, leads to a layer (2) communicating co-processor structure, that is, a single Jacobi processor.

2. The layers in the hierarchy

As said before, level (0) functions are the (applicative state transition) functions in the dependence graph of the Jacobi algorithm. The dependence analysis as well as the generation of the dependence graph is performed by a tool (called **HiPars** [8]). For example, the input to **HiPars** for the adaptive QR algorithm is (part of) the following nested loop program.

Where the functions vecOP and rotOP are both ECTs $M(\alpha, \alpha, \varphi)$, conditional and unconditional, respectively, with $M(\varphi) = J(\varphi)$. **HiPars** returns a reduced dependence graph as shown in figure 7.

The nodes ND_{20} and ND_{45} are the index domains supporting the functions *vecOP* and *rotOP*, respectively. The arcs are inter and intra node dependencies, see [8]. These functions are implemented using the shift-and-add core operations at level (−2). They may be full angle range Cordics [9] [10], or restricted angle range (inexpensive) μ-rotations [12] [11]. A ECT generator - called **Bangles** is available with which the shift coefficients for the look-up table at level (−2) are computed (given type of ECT, angle range and required accuracy or cost bound). The specification of level (−2) is as shown in table 1.

Similar tables could be given for the other levels as well. They are omitted for lack of space. The 'Port Map' defines the relation between the port type

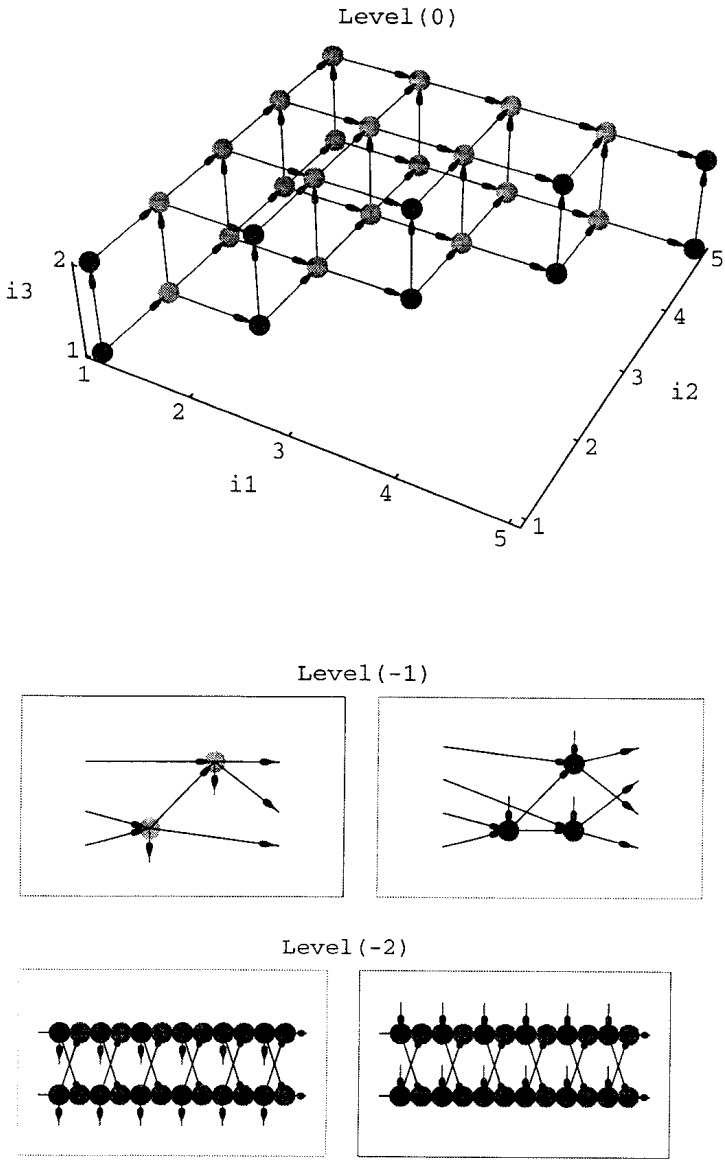

Figure 4. Hierarchical decomposition of the level 0 graph.

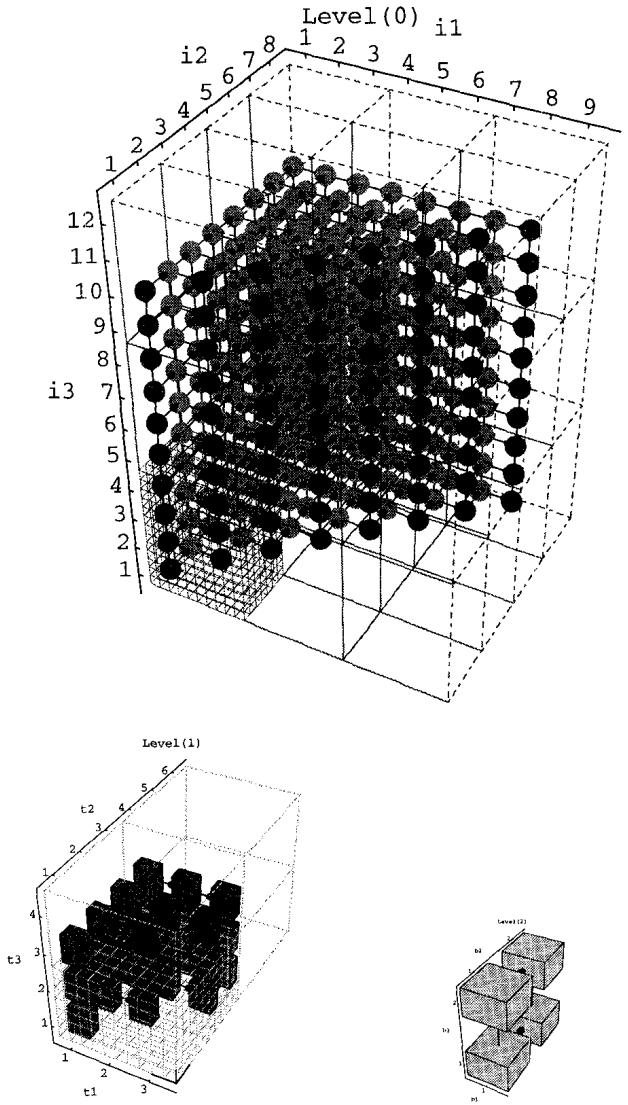

Figure 5. Hierarchical partitioning of the level 0 graph.

		Notes
Level	-2	
Function Set	$\mathcal{F}^{(-2)}$	$F_1^{(-2)} = \text{add}$ $F_2^{(-2)} = \text{shift}$
generated Control Vector	$\alpha^{(-2)}, \beta^{-2} \in \mathbb{Z}^+$	loop control shift factor
Composed Set	$\mathcal{F}^{(-1)}$	$\begin{bmatrix} r \\ 0 \end{bmatrix} \leftarrow M(\varphi) \begin{bmatrix} x \\ y \end{bmatrix}$ $\begin{bmatrix} x' \\ y' \end{bmatrix} \leftarrow M(\varphi) \begin{bmatrix} x \\ y \end{bmatrix}$ $M(\varphi) \in \{J(\varphi), G(\varphi), H(\varphi), N(\varphi)\}$
Port Map	$\mathcal{P}^{(-1)}$	Identity map
Assignment	Application specific	add $\mapsto$ full adder, shift $\mapsto$ barrel shifter
Scheduling	in Order	Scheduling follows the lexicographical ordering of the computation graph (table evaluation)
Input Control	$\alpha^{(-1)} \in \mathbb{Z}^+$	scalar pointer in $\mathcal{F}^{(-1)}$
Programmability	$\{\phi_i\}$ $K-$ factor	provided as sequence of alternating shift and add operations

Table 1. Level -2 functional description (Cordic core)

```
function vecOP;
function rotOP;
function Pass;
parameter Kmax 1 128;
parameter N 8 48;
for k = 0 : 1 : Kmax - 1,
  for d = 0 : 1 : N - 1,
    [R(d,d),X(k,d),theta(d)] = vecOP(R(d,d),X(k,d));
    for c = d+1 : 1 : N - 1,
      [R(d,c),X(k,c)] = rotOP(R(d,c),X(k,c),theta(d));
    end
  end
end
```

Figure 6. adaptive QR algorithm in matlab NLP form.

of a member , say $f_j^{(i)}$, of the set $F^{(0)}$, and the port type of the 'physical' node into which the function is mapped. For example, the input and output of the conditional rotation $J(\varphi)$ is a 2D and 3D vector, respectively, whereas these are both 3D for the unconditional rotation $J(\varphi)$. However, both may be mapped into a physical node having only scalar input and output ports. As a result, the vector valued arguments and results of the functions have to be converted to input and output streams of scalars as part of the mapping of the functions into a physical node.

The compound level implements a generic Jacobi step which we define here to be

$$X' \leftarrow A(\alpha, \beta, \theta)\, X\, B^T(\gamma, \delta, \phi)$$

where all matrices are of dimension 2. The corresponding (irregular) computation graph is a network of communicating ECT nodes. The distinction between conditional and unconditional ECTs is made through a control variable that is received by the node to select between them. In addition to these functions there are some register handling functions for initializing or copying memory cells. Programmability is through program selection from a program memory. This program contains read and write instructions to realize the

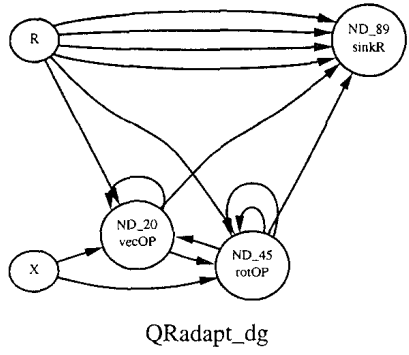

Figure 7. Reduced dependence graph for the adaptive QR algorithm.

flow in the graph as well as selection instructions which are sent one level down to select the node functions. This in turn causes fetching of instructions needed at the lowest level to compose these functions. The functionality of the nodes in the dependence graph (level (0)) is given in the form of a sequential program. The conversion of this program to a program that the interpreter can understand can be implemented in e.g., `Mathematica`.

It will by now be clear that at levels having an index $i \leq 0$, functions are stored in the form of a set of instructions and are selected by the control vector $\alpha_j^{(i)}$. Strategy changes, however, when considering layers with index $i > 0$ as they correspond to graphs that are piecewise regular. The control vectors, then, have a parameter setting nature. For example, the dependence graph of a Jacoby algorithm - defined at level (0) - may be partitioned using parallelepiped shaped tiles. The tool that does this outputs a (reduced) tile graph [7] as well as the specification of a single tile, again a reduced dependence graph of it. This tile may be parametrized with respect to its size. The clustering or folding of the tile requires a sequential schedule which is expressed in the form of a nested loop program. This program is obtained from the dependence graph of the tile by reversing the procedure that generates dependence graphs from nested loop programs. The parametrized schedule program is stored on layer (1) of the Jacobi co-processor and the control vector $\alpha^{(0)}$ carries the parameter values.

3. Simulation of the adaptive QR Jacobi array

In order not to overburden the section we define a 'mini' Jacobi processor which consists of only one Jacobi co-processor and which processes only real data. Moreover, level (-2) has been omitted so that the lowest level is the ECT level om which the ECTs are functions, not graphs. As test vehicle we considered is the time-adaptive QR algorithm with real values only. In nowadays wireless applications (GSM) a typical example uses up to ten antennas ($N = 10$) and requires a data throughput rate of 275 k vectors per second. A first investigation of the algorithm shows that each vector comprises $\frac{10 \cdot 11}{2} = 55$ Cordic operations. Assuming we use a serial Cordic, as is the case in the Jacobi processor, a clock rate of 40 MHz and each Cordic operation takes 32 cycles then we need at least

$$\frac{40 \text{ MHz}}{\frac{55}{p} 32} = 275 \text{ kHz}$$

so $p = 12.1$ processors. In a triangular array this yields

$$\frac{p_i (p_i + 1)}{2} \geq p$$

or $p_i = 5 > 4.4$ and $p = 15$. This choice allows a duty cycle of

$$\frac{40 \text{ MHz}}{\frac{55}{15} 32\, d} = 275 \text{ kHz}$$

$d = 1.24$ or ≈ 40 cycles per Cordic, that includes memory access.

It will turn out (figure 8) that we need on average 47 cycles per Cordic or a duty cycle of $d = 1.47$. The throughput is thus

$$\frac{40 \text{ MHz}}{\lceil \frac{47}{15} \rceil 55} = 210 \text{ kHz}$$

Figure 8 shows the respective output times[2] for each x vector entry x_i $(1 \leq i \leq 10)$. It takes up to 188 units of time before a next value of a vector entry x_i is produced.

The various parameter setting in this particular simulation have been collected in table 2. A plot of the load balance monitor is shown in figure 9 in terms of the time profiles of the 15 'mini' Jacobi processors.

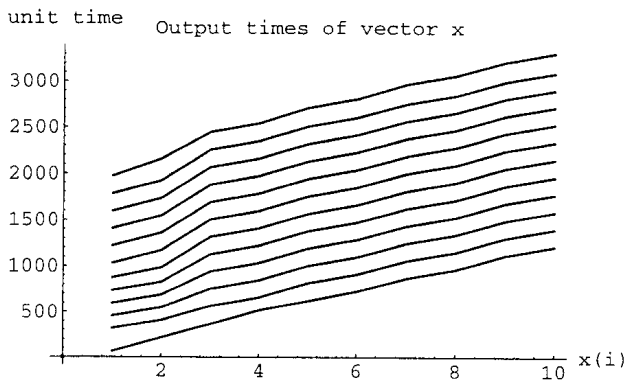

Figure 8. Out put times of entries $x_{i,k}$ of the processed input vectors x_k in adaptive QR algorithm.

level -1 (Cordic)	Vectoring 32	Rotation 32
level 0 (Compound)	control FiFo cap. 2	data FiFo cap. 5
level 1(a) (Cluster)	control FiFo cap. 3	data FiFo cap. 6
level 1(b) (Cluster)	control FiFo cap. 20	

Table 2. Parameter settings in the adaptive QR simulation on the triangular array of 5×5 Jacobi processors

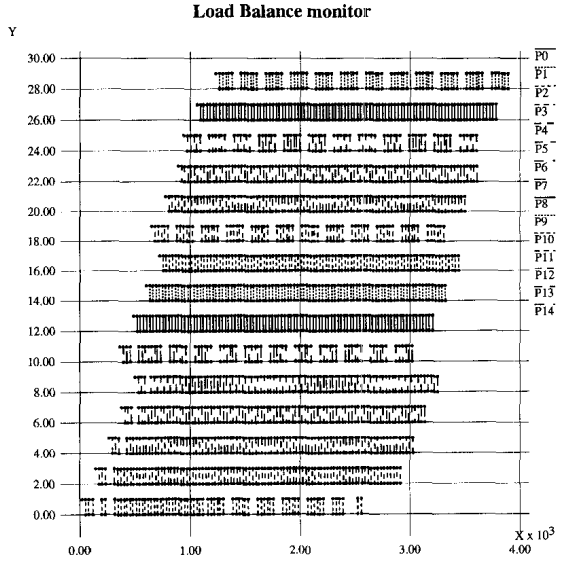

Figure 9. Activity monitoring of the Cordic cores in each of the Jacobi processors in figure 13.

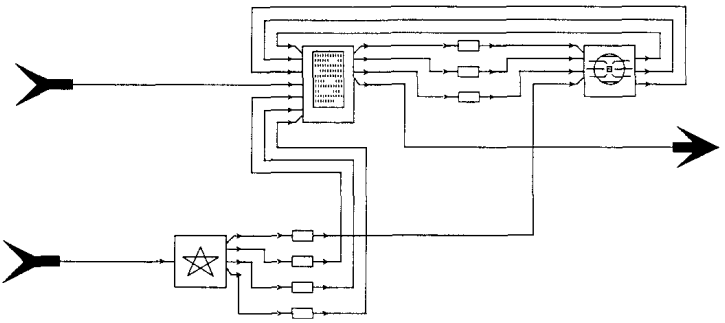

Figure 10. The compound layer of the 'mini' Jacobi processor.

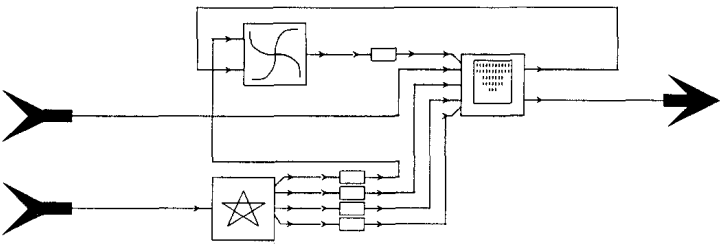

Figure 11. The cluster layer of the 'mini' Jacobi processor.

The simulation as a whole was performed in Ptolemy. The various layers are shown in figures 10 (Compound node built on the Cordic core), 11 (Cluster node), and 12 (The 'mini' Jacobi processor). The triangular array configuration of the test vehicle is shown in figure 13.

4. Conclusion

The processor, architecture and compiling strategy presented here is based on a hierarchical specification and description of piecewise regular algorithms and architectures. Starting from a piecewise regular dependence graph, the function nodes are refined hierarchically stepping down the hierarchy and

[2]These are the time instants at which the entries of incoming vectors are nulled. In steady state operation of the system, they are proportional to the time instants at which new vector samples are taken in.

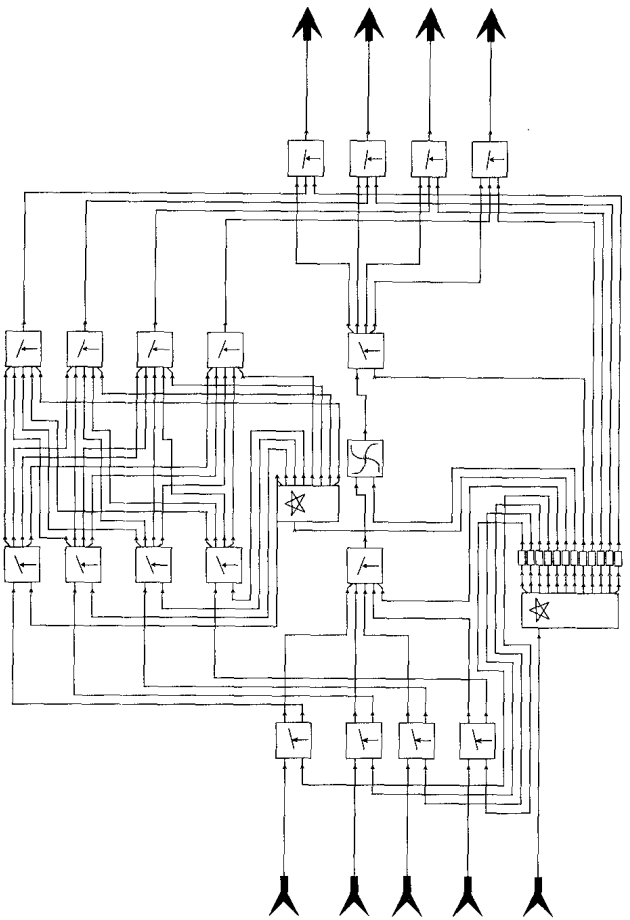

Figure 12. The 'mini' Jacobi processor, the data path and cluster function are visible.

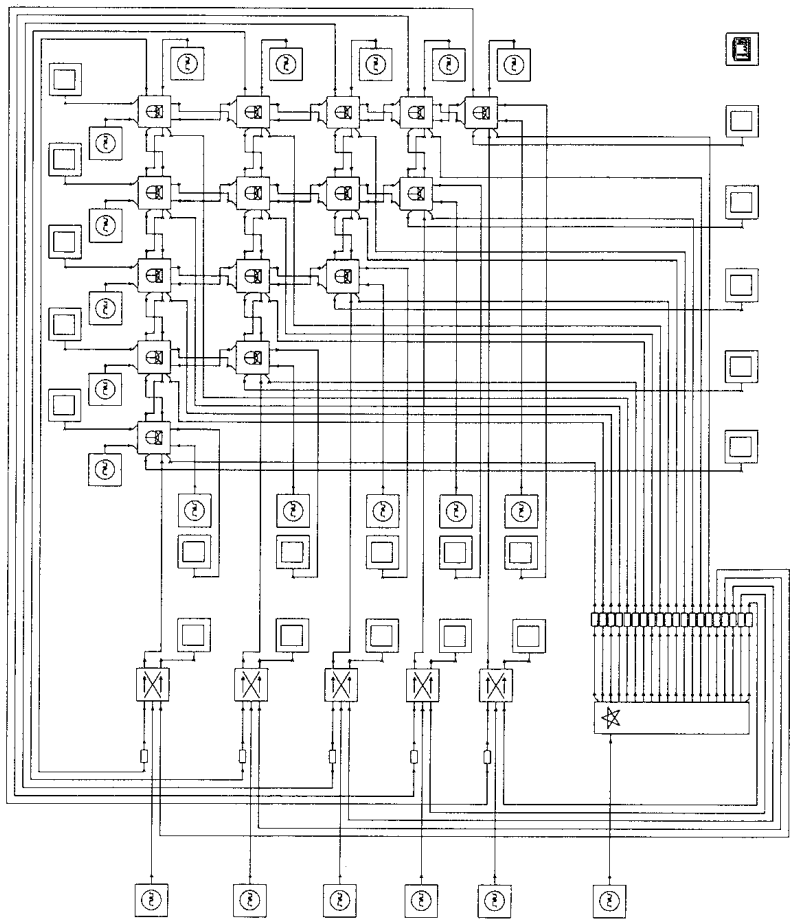

Figure 13. A Tri array of fifteen 'mini' Jacobi processors.

penetrating eventually into the domain of irregular flow graphs. Partitioning (tiling) the dependence graph takes us up in the hierarchy until we reach the system level. The example that we provided has been overly simplified - the size of the antenna array has been rather small and so has been the size of the cluster tile; moreover the Jacobi processor has been taken to be identical to the Jacobi-co-processor as a result of which computations inside it are all serially ordered - a result of which the emerging array turns out not to be very efficient. In a more realistic scenario better performance would not be difficult to obtain. On the other hand, overhead cannot be avoided as the architecture is family specific. However, an essential aspect of the approach is that the compiler makes uses of tools from a design system so that the capabilities of it will grow with those of the system. This facilitates achieving the goal of providing multi-processor structures with which Jacobi algorithms could be quickly prototyped. For an application specific Jacobi algorithm, specific architectures and processors can be designed which will most likely be quite different from the Jacobi arrays and processors, respectively. For example, the same adaptive QR algorithm (processing complex data) has been implemented in a single pipelined complex Cordic (three real parallel Cordic pipelines) [15] which could be easily mapped into a single chip. With the Jacobi processor, however, the behavior and performance of many Jacobi algorithms can be evaluated quickly by interconnecting Jacobi processors in a multi-processor configuration. Some of these algorithms are complicated, yet the design/compile toolbox contains several transformation tools, including algorithmic transformation tools [16] to render the mapping onto the Jacobi array sufficiently efficient for prototyping purposes.

5. ACKNOWLEDGEMENT

This research was supported in part by the Commission of the EU under contract ESPRIT BRA 6632 (NaNa2) and in part by the National Technology Foundation under contract DEL55.3941.

References

[1] A. Abnous and J. Rabaey, "Low-Power Application-Specific Processors,".

[2] A.-J. van der Veen, Ed F. Deprettere, and A. Lee Swindlehurst, "Subspace-Based Signal Analysis Using Singular Value Decomposition," *Proceedings of the IEEE*, **81**(9):1277–1308, 1993.

[3] J. Götze and A.J. van der Veen, "On-line Subspace Estimation using a Schur-type Method," *IEEE Trans. Signal Processing*, October 1995.

[4] Filiep Vanpoucke, Marc Moonen, and Ed F. Deprettere, "Stable Jacobi SVD Updating by Factorization of the Orthogonal M atrix," In Marc Moonen and Bart De Moor, editors, *SVD and Signal Processing, III*, pp. 267–276. Kluwer Academic Publishers, Dordrecht, 1995.

[5] P. Kapteijn, H. van Dijk, and E. Deprettere, "Controling the critical path in time adaptive QR^{-1} recursions," In *Proc. IEEE Workshop on VLSI Signal Processing*, volume VII, pp. 326–335, 1994.

[6] H.W. van Dijk, G.J. Hekstra, and E.F. Deprettere, "Scalable parallel processor array for Jacobi-type matrix comput ations," *Integration, the VLSI journal*, **20**:41–61, 1995.

[7] Ed Deprettere, Peter Held, and Paul Wielage, "Model and Methods for Regular Array Design," *Int. J. of High Speed Electronics, Special issue on Massively Parallel Computing-Part II*, **4**(2):133–201, 1993.

[8] Peter Held, *"Functional Design of Data-Flow Networks,"* PhD thesis, Dept. EE, Delft University of Technology, May 1986.

[9] A. de Lange and Ed F. Deprettere, "Design and Implementation of a Floating-point Quasi-Systolic General Purpose Cordic Rotor for high-rate Data and Signal Processing," In *Proceedings 10th IEEE Symp. Computer Arithmetic*, pp. 272–281, Grenoble, France, June 26-28 1991.

[10] Gerben J. Hekstra and Ed F. Deprettere, "Floating Point Cordic," In *Proceedings ARITH 11*, Windsor, Ontario, June 30 - July 2, 1993 1993.

[11] Ed F. Deprettere, Gerben Hekstra, and Richard Heusdens, "Fast VLSI Overlapped Transform Kernel," In *1995 IEEE Workshop on VLSI Signal Processing*, volume VLSI Signal Processing VIII, pp. 287–302, Osaka, Japan, October 1995.

[12] J. Götze and G. Hekstra, "An algorithm and architecture based on orthonormal μ-rottions for computing the symmetric EVD," *Integration, the VLSI Journal*, **20**:21–39, 1995.

[13] L. Thiele, "On the design of piecewise regular processor arrays," In *Proc. IEEE Symp. on Circuits and Systems*, pp. 2239–2242, Portland, 1989.

[14] S. Haykin, *"Adaptive Filter Theory,"* Englewood Cliffs, NJ: Prentice Hall, 1992.

[15] P. Kapteijn, E. Deprettere, L. Timmoneri, and A. Farina, "Implementation of the recursive QR algorithm on a 2x2 CORDIC te stboard: a case study for radar applications," In *Proceedings 25th European Microwave Conference*, pp. 500–505, Bolognia, September 1995.

[16] Hylke W. van Dijk and Ed F. Deprettere, "Transformational Reasoning on Time-Adaptive Jacobi Type Algorit hms," In Marc Moonen and Bart De Moor, editors, *SVD and Signal Processing, III*, pp. 277–286. Kluwer Academic Publishers, Dordrecht, 1995.