A High-Level Simulator for the Performance Analysis of Dataflow Implementations of Jacobi algorithms

Daniël van Loenen and Bart Kienhuis
Delft University of Technology, Mekelweg 4,
2628 CD, Delft, The Netherlands
email: kienhuis@cas.et.tudelft.nl

Abstract—**Objective of the Jacobium project [1] is to make a weakly programmable and scalable processor that is tailored to a specific application domain in signal processing. In designing of such processor, we are faced with the problem that there is a large number of design-parameters of which it is unknown what values they should have. Hence we like to perform a design-space exploration to get an idea of what combinations of design-parameters result in the best performing configuration for the set of Jacobi-algorithms. In this paper we demonstrate how a high-level, cycle-accurate, and functional correct, simulator can be constructed to perform this design space exploration, using the method advocated in [2], [3]. We also show in what kind of environment the simulator needs to be embedded. We conclude this paper by drawing up some lessons learned from constructing the simulator environment.**

I. Introduction

OBJECTIVE of the Jacobium project [1] is to make a weakly programmable and scalable processor that is tailored to a specific application domain in signal processing. The applications show up in array processing [4] and time series analysis [5] problems. In these problems, the objective is to detect, recover or separate signals, or to estimate signal or system model parameters.

A characteristic and common feature of the algorithms we are referring to here is that they perform matrix computations, in particular matrix factorizations [6], which typically account for a heavy computational payload when it comes to implementing these algorithms in signal processors.

Examples of such algorithms include the *QR* and *SVD* matrix factorizations [6] which are, indeed, frequently encountered in modern high-resolution signal processing applications. A numerically and structurally appealing implementation approach is to cast the factorization into a so-called Jacobi-algorithms. Adaptive QR decomposition is a good example of such Jacobi-algorithm. It uses *vectorizations* and *rotations* in order to compute the decomposition. Although both vectorizations and rotations can be implemented in commercially available signal processors, the performance may not be satisfactory for two reasons:
1. The conventional multiply/add arithmetic that these processors suppose is neither efficient nor optimal for performing vectorizations/rotations.
2. The processor's architecture is not suitable for the inherent dataflow-type nature of the algorithm's structure, as is revealed by the so-called dependence graph representation. For the adaptive QR decomposition, a snapshot is given of the dependence graph as shown in Figure 1.

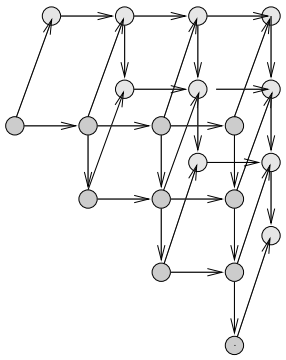

Fig. 1. Two plane of the dependence graph of adaptive QR

Striking features revealed by a dependence graph representation are regularity, locality, and concurrency. These features are, to a large extent, characteristic for all Jacobi-algorithms. It thus makes sense to speak of *class of Jacobi-algorithms*, to refer to class properties, and to address the problem of specifying and designing a processor which is specific for that class. The latter is exactly what the Jacobium project [1] is aiming at: designing a weakly program-

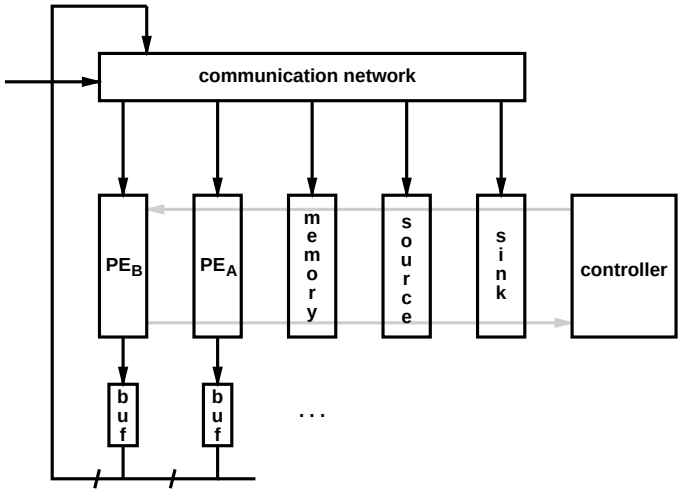

Fig. 2. Architecture Template of the Jacobi Domain-specific Processor

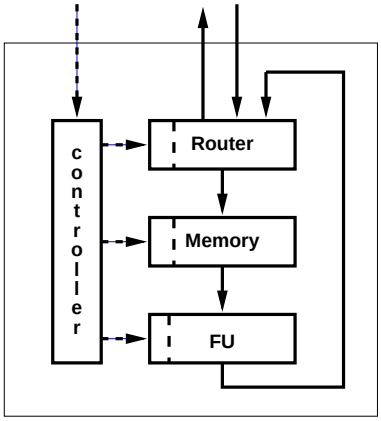

Fig. 3. Model of a Processing Element (PE)

mable and scalable processor, suitable for the efficient execution of some sub-set of the class of Jacobi-algorithms. These class properties make it feasible to make a domain-specific processor that utilizes parallelism, and has a small instruction set. An example of an abstract model of the processor is given as an architecture template in Figure 2. This architecture template consists of in parallel operating *processing elements* (PEs) that can communicate with each other via a *communication network*. An important part of the architecture is the Processing Element. A model of such PE is shown in Figure 3. A *PE* consists of four elements:

1. a *controller* which contains a control program and instructs the other elements by means of compact commands,
2. a *Router* which directs incoming and outgoing data,
3. a *Memory* which acts as a re-order buffer for the Functional Unit,
4. a *Functional Unit* which performs the vectorizations and rotations functions onto the data.

In designing the Jacobi-processor, we are faced with the problem that there is a large number of design-parameters of which it is unknown what values they should have. This includes aspects like the size of the processor's memory or the way the PEs are best interconnected. Hence we like to perform a design-space exploration to get an idea of what combinations of design-parameters result in the best performing configuration for the set of Jacobi-algorithms.

Given the characteristic features of the Jacobi-algorithm, notably the regularity and locality of their dependence graph, one may be tempted to think that a qualitative analysis would be a feasible approach to the performance analysis of implemented Jacobi-algorithms and the hardware on which they are executed. Indeed, there exists a comprehensive literature dealing with so-called piecewise regular algorithms, to which the Jacobi-algorithms belong, ranging from dependency analysis and dependence graph transformations, to partitioning strategies and scheduling techniques. However useful these methods may be, they usually aim at converting a single piecewise regular architecture to a single piecewise regular flow-graph which is, then, considered *the* architecture for *that* algorithm. This is a different objective than ours, which is to determine the optimal set of parameters for the given Jacobium-processor architecture template and *set* of Jacobi-algorithms. Surely, these qualitative methods may play a role in the design-space exploration, yet a quantitative analysis of the performance of the architecture's execution of these algorithms is indispensable for the determination of the optimal values of the parameters in both the architecture and the algorithms.

In a design-space exploration we want to measure the performance of mappings of algorithms from the Jacobium-class onto architecture instances of the Jacobi-processor. The environment for performing quantitative analysis that can be used in a design-space exploration, is shown in Figure 4. This environment should also provide feedback to the design efforts in hardware, the construction of the mapping (also referred to as a compiler) and the way the algorithms are written. More concretely: a simulator is needed that benchmarks the system's possible configurations and mappings and gives a quantitative analysis of its performance in terms of:

- the speed of the system is (in a particular configuration) in terms of total time needed to perform a

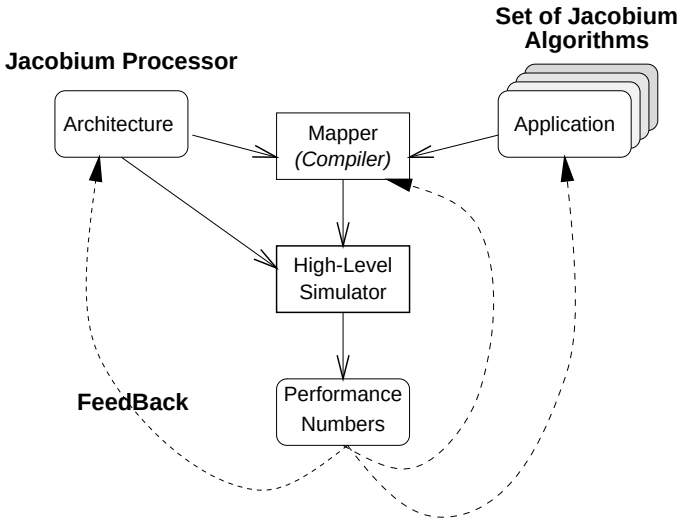

Fig. 4. Environment for Performing Quantitative Analysis

practical computation

- the size of (in bits) the memory needs to be in that configuration
- how it compares to other configurations

The remaining part of this paper is organized as follows. In section II we explain what kind of simulator we need and what method we should use. In section III we explain how we can construct high-level simulators with the proposed method. In section IV we describe the construction of the simulator of the Jacobi-processor. The simulator is embedded in an environment (based on Figure 4) what is explained in section V. In section VI we conclude this paper by presenting some lessons learned when constructing the environment and draw up some conclusions.

II. what kind of simulator do we need?

We are interested in the performance of the system in different configurations (i.e. combinations of hardware and control programs generated by the mapper), but is still accurate enough to compare different configurations. We want a model that has as few details as possible. We are *not* interested in hardware details such as the timing behavior of, say, a multiplier component; to perform design-space explorations, this kind of detail is not necessary. High-level modeling results in a simpler executable model, which translates into a shorter computer run-time. This is a more than useful feature because of the large number of simulations we may want to perform. Another important result is that a less detailed model is easier to change. Current practice is to either use VHDL or 'C' to make such a simulator.

Using VHDL, means modeling individual subsystems at a level of detail that is generally not necessary for design-space exploration. It is therefore too slow in terms of both changeability and simulator run time. Making a simulator using 'C' generally produces a simulator that runs much faster than one made in VHDL. In 'C' a high-level model can be constructed that can execute very time-efficiently. But since 'C' itself does not support parallel processes, a programmer has to construct a schedule. This schedule may cost the programmer a lot of time to figure out. When a different system configuration is to be simulated, a new schedule is needed. Such a simulator will run reasonably fast compared to one made in VHDL, but the effort required to change aspects of an architecture is still to time consuming.

A different approach has been suggested by Bart Kienhuis [2], [3]. This approach uses C++ to construct instances of the architecture and a high-level, multi-threading scheduler, to implemented the notion of time and parallelism. The method he proposes enables us to make a simulator that are both fast in terms of computer run-time, and the time needed to change the configuration. The object-oriented nature of C++ suits it well for modeling different levels of hierarchy within the system and the multi-threading scheduler removes the burden to schedule the architecture.

From the PAMELA [7] work of Arjan van Gemund, we know that it is enough to model *resource contention* and *mutual exclusivity* to determine the performance of a parallel system. This makes it possible for us to make an abstract model of the system, without having to go into much detail. Only three primitives are required to model these two aspects and they are included in the PAMELA run-time library [8]. It provides parallel running *processes*, *semaphores* and the notion of *time* and is the multi-threading scheduler used.

III. Principles of constructing the simulator

To show how the C++ programming language and the Run-time library can be used to describe a high-level, cycle-accurate executable PE model, as given in Figure 3, we first look at a very simple model of this PE as given in Figure 5. The *structure* of this simple PE model is constructed using C++ objects. Thus each element of the PE, the Router, the Memory, and the Functional Unit, is modeled as a C++ object. Within each object a PAMELA process is instantiated. These processes can run in parallel. The ele-

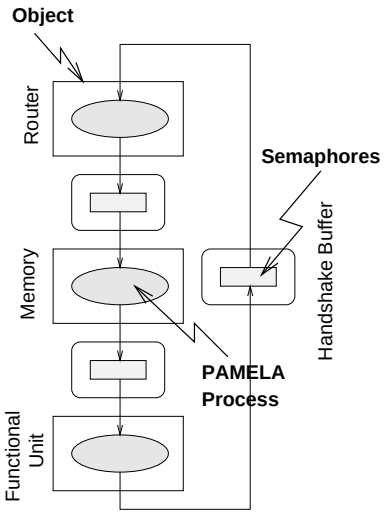

Fig. 5. A Simple Executable PE Model Using Processes, Semaphores and C++-Objects

ments in figure 5 are connected with each other via handshaked buffers. These buffers are also model as C++ object, but contains no processes but only semaphores. The simulator for this simple PE model therefore has three processes, running in parallel, pumping data to each other using the handshaked buffers. Next we look in more detail how a Handshake buffer and a Functional Unit is modeled using respectively semaphores and processes.

A. Semaphores

The handshaked buffer is an unidirectional buffer which can contain one single data token. This buffer can be placed between two processes. When the first process writes an element in the buffer, the second process can read the element from the buffer. If the buffer is full, the first process blocks while trying to write data, and will only continue after the second process reads the buffer, thereby clearing it. When the buffer is empty, the opposite is true. Semaphores are used to control the blocking of the processes. Semaphores have an integer value that can be changed using the pam_P construct to decrement a semaphore by one or pam_V to increment a semaphore by one. If the value of a semaphore becomes zero (e.g. using a pam_P), it will block a process. An example of how semaphores can be used in a handshake buffer is given in Figure 6. A semaphore *room* keeps track of the available room in the buffer, and semaphore *data* indicates whether data is present in the buffer.

```
Token* HandShakeBuffer::read()
{
   pam_P(data); //Buffer contains a data element?
   Token* temp = new Token();
   temp = token;
   pam_V(room); //indicate there is room in the buffer
   return token; //return the contents of the buffer
}
void HandShakeBuffer::write(Token* aToken)
{
   pam_P(room);  //is there room to write an element?
   token = aToken;//store the data element
   pam_V(data);  //indicate there is data in the buffer
}
```

Fig. 6. A Example of How Semaphores and C++ are Used to Model a Handshake Buffer

B. Processes

Only Processes can make progress in time. This characteristic is used to model the timing behavior of an object within a system (e.g. the simple PE model). In Figure 7 an example is given of how a process is created for the Functional Unit in the simple PE model.

```
FunctionalUnit::FunctionalUnit(HandShakeBuffer *in,
                               HandShakeBuffer *out)
{ // Constructor of a Functional Element
   inBuffer  = in;   // Store the In/Out buffer
   outBuffer = out;  // within the Object
   process = pam_fork(FU,process_fu,this);
}
void process_fu(void)
{
   FunctionalUnit* fu = (FunctionalUnit*) pam_args(pam_me());
   HandShakeBuffer *inBuffer  = fu->in;
   HandShakeBuffer *outBuffer = fu->out;
   while(1) {
      Token *token = inBuffer->read();
      pam_delay(1);
      outBuffer->write(token);
   }
}
```

Fig. 7. An Example of How a Process is Create for a FU and How it Interacts with the FU-Object

In the example, we first show the constructor of the FU-element. The constructor has two arguments, the two buffers that connect the FU with the other elements e.g. the Router and the Memory. These buffers are stored within the data-structure of the FU-object. Next the process of the FU is instantiated , based on the function `process_fu`, using the `pam_fork` statement. One argument is passed to the process, the *this* pointer which represents the data- structure of the FU-object. Next the process function `process_fu` is given in the example. This process first retrieves the data-structure from the FU-object that instantiated the process using the `pam_args` and `pam_me` statements. Subsequently the process obtains the correct

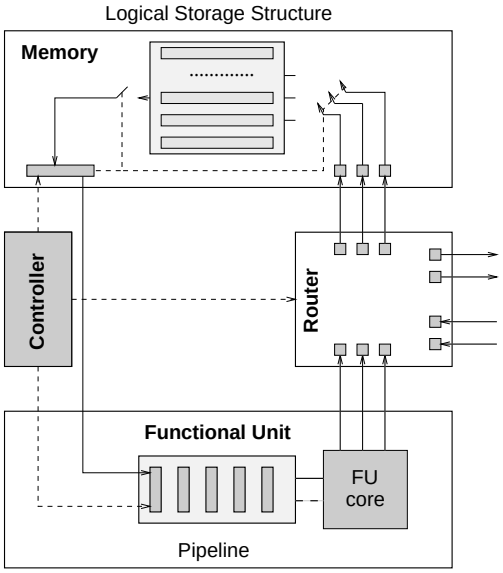

Fig. 8. The complete PE model

in- and output buffers from the FU-object. At this stage the complete structure of the FU-element is constructed including processes and buffers and the FU-process enters an infinite while loop. Within this while-loop, the process first tries to read, at t=0, a token from the input buffer of which the structure is already shown in Figure 6. If a token is read, the process is delayed 1 cycle and at t=1, the process tries to write the token to the output-buffer. At t=1, the process again tries to read a token. If a token was not available on the input buffer at t=0, the FU-process was blocked for a certain amount of time, caused by condition synchronization, until the memory-process writes a token.

IV. The PE simulator

The principles described in the previous section are used to construct an executable model of the complete PE-model, as shown in Figure 8. Within the figure, three elements have been added, a controller, a pipelined-structure in the Functional Unit and an array of buffers within the Memory. The remaining shaded squares represent parallel processes. The Router, for example, is now modeled with 10 processes instead of one in the simple PE model. Each arrowed-line in the figure represents a Handshake buffer. Now we explain, in more detail the various elements of the complete PE model. Data that flows through the machine, is represented as *tokens*. Tokens can contain additional information, like for example, to which PE it should be send. This way streams of data can be addressed to particular PEs.

The *Router* handles the interaction with the outside of a PE. It can receive tokens from both the outside or from its own FU. A token coming from the outside enters the Router and if the token has the right header (i.e. the destination address), the token will go to the Memory. Otherwise the token will be send back to the outside. A similar procedure is used when a token comes in from the *FU*.

The *Memory* element contains an array of *Logical Storage Structures* buffers (see [9] for more information). An LSS-buffer is an in-order write, out-of-order read buffer. Using these LSS-buffers, streams of data can be re-ordered, as is required by some of the Jacobi-class algorithms like MVDR [10]. Within the Memory, there are also switches to the array of LSS-buffers, both at the input as well as the output side. Using these switches, the Controller can determine in which LSS-buffer data should be written or from which LSS-buffer data should be read.

In the *Functional Unit* of the Jacobi-processor data is processed using a *pipelined Cordic*, where in each stage of the pipeline, part of a vectorization/rotation is executed. The FU is modeled as consisting of two parts, an asynchronous pipeline and an FU-core. The asynchronous pipeline consists of handshaked slices (each of which is a parallel process) that only pass the data through, followed by a FU-core that performs the combined functions of the entire pipeline. This way we can also perform a functional correct simulation.

The *Controller* contains a *control program*. In each cycle, it translates this program into *command words* that are written into the control-lines (dashed-lines in Figure 8) which connects the Controller with the rest of the elements in the PE. Through these control-lines, the Controller tells, for example, the *Memory* where to fetch its data from or which operation the FU has to perform on its data and where to send its output to (i.e. how to change the header of the token).

The Controller is the connection between the hardware design effort of the PE and the mapping effort for the PE. The way the FU, Memory, and Router are modeled, directly influence the kind of commands the Controller may issue. An algorithm, like adaptive QR, must be expressed in these commands in order to be executed on the PE. By making the commands more sophisticated, the hardware effort increases to support these more powerful commands, but the mapping might become easier. Or the other way around, we could create simpler commands that are easier to implement in hardware but make it harder to map algorithms. Finding the right balance between the hard-

ware complexity of the PE and the mapping complexity of the PE is something we want to find out. Because the simulator is modeled at a high-level, changes can be incorporated quicker and more easily.

The PE is only one element in the architecture template model shown in Figure 2. Therefore we also modeled the *communication structure* and the *controller* to get an executable model of the Jacobi-processor. The Jacobi-processor simulator is parameterized to support a limited flexibility. The parameters of the simulator are set via a text file that contains information about different architectures in terms of interconnection patterns, although we can only create a linear array of PEs at this moment. It also contains information about the number of PEs used and for each PE it contains information about, for example, the number of pipeline-slices to be used in the pipeline of the FU, the number of LSS-buffers used in the Memory or timing parameters for the various elements in the PE.

V. Performing a Simulation

The Jacobi-processor simulator is created to be used in the environment as shown in Figure 4. How we have implemented this environment is shown in Figure 9.

A. Simulation Environment

By selecting *architecture parameters*, we can select a specific instance of the architecture template of the Jacobi-processor. Using the technique of C++ and the PAMELA Run-time Library we can construct a high-level, cycle-accurate, functional correct, execution model (e.g. the *architecture simulator*) of this Jacobi-processor instance. To execute Jacobi-class algorithms on the Jacobi-processor instance, we need a *mapper*. Although not discussed in this paper (see [11] for more information), we used a mapper, written in MATLAB that takes as input a Jacobi-class algorithm, and the architecture parameters. The mapper generates a *control program* for the architecture simulator. This control program is stored in the controller of the various PEs used. Now we can start the execution of the architecture simulator. The architecture simulator will generate all kinds of performance output. To get this output the PEs have been instrumented with probes. These probes measure during the execution, contentions, utilizations, and throughputs. These number can be used to analyze the architecture instance and to generate feedback to the architecture design, mapping strategies or

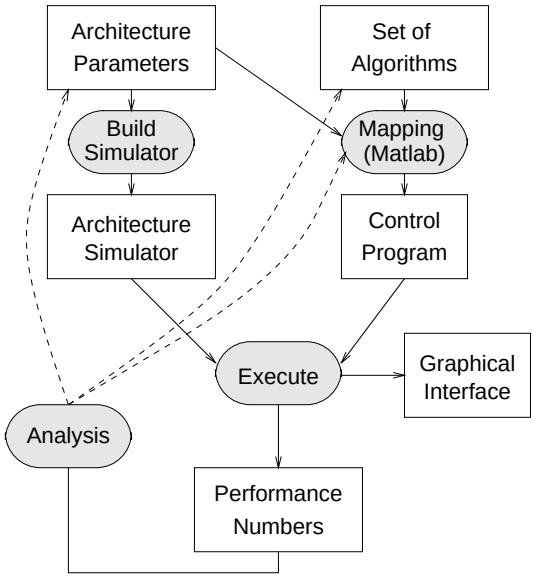

Fig. 9. The Simulation Environment Used

algorithm-design efforts. During execution of the architecture simulator, a *graphical interface* is displayed that shows how data flows through the architecture instance. This is mainly used to debug the architecture simulator, not a real luxuriance when building parallel system simulators! Moreover, *functional simulations* are used for validation of the correct working of the PEs and mapping.

B. An experiment

A simple experiment has been performed, were we aim at getting the following performance metrics:

- operations (vectorizations + rotations) per clock cycle, denoted as *parallelism*
- memory required, or the maximum number of data elements that have to be stored at one time in the LSS-buffer structure, denoted as *memory size*

All experiments were performed using a QR algorithm. We were interested in what these performance metrics are for various problem sizes, and for a different number of *PEs*. In each experiment, the total number of planes in a dependence graph (see Figure 1) that has to be executed is the same: 16 planes. The results of the experiment are shown in table I.

Examining these results, we see that when we use more PEs per problem, the total execution time (clock-cycles) gets shorter, but this does not scale in a linear way. Thus the QR algorithm is not fully exploiting the amount hardware available. When we examine the mapping generated, we can see that there is a large proportion of the tokens are transported from one PE to its right-hand neighbor PE, putting a lot

TABLE I
SOME SIMULATION RESULTS

Number Antenna's	PEs	Parallelism Op./Cycle	Memory Size Data words
8	1	0.90	174
8	8	1.23	71
16	1	0.94	505
16	8	3.00	271
16	16	3.26	270

of load on the communication structure. Hence we can try to make a communication structure that has a higher bandwidth, or we could try to map the QR algorithm in such a way that data is kept more locally to a PE.

VI. Lessons Learned & Conclusions

In the construction of the high-level simulator of a Jacobi-processor and the environment the simulator is embedded in, we have learned the following lessons and can draw up the following conclusions.

- The method advocated in [2], [3], to use C++ and the PAMELA Run-time library, is used successfully in the building of a high-level, cycle-accurate, functional-correct, execution model of the Jacobi-processor that is still (limited) reconfigurable and can be used in a design space exploration.
- We had to set up the environment as shown in Figure 9 early in the project. Therefore we got confronted early with many aspects of the design of the PEs that would normally appear only much later in the design. Furthermore we had to include the mapping issues early on. This way the construction of the architecture and mapping goes hand-in-hand which is an absolute necessity when constructing programmable architectures. By the time the architecture is build, the mapping is also addressed.
- Looking back at the level of detail of the simulator, an even more abstract model could have been constructed. Because the construction in hardware started at the same time as we started the construction of the simulator, we were biased to a particular implementation and already concentrating to much on details. It would have been better to start the hardware effort after the first explorations have been performed.
- In the construction of the simulator, the elements are to much intertwined. It is therefore difficult to take out an element, say the Memory element in the PE, and to replace it with an other element. This seriously limits the kind of explorations we can perform.
- The Jacobi-project is a multi-disciplinary project. Improvements in the Jacobi-processor can be achieved by either changing the hardware, the way the Jacobi-algorithms are written, or the mapping strategies used. The environment of Figure 9 captures these disciplines early. The construction and execution of the simulator allowed us to give early feedback to the other disciplines and to withdraw valuable input from them to improve the overall processor design.

VII. Acknowledgment

The authors would like to thank all members of the Jacobi-processor project for the many fruitful discussions we had with each of them showing their side of the story in designing the Jacobi-processor.

References

[1] Edwin Rijpkema, Gerben Hekstra, Ed Deprettere, Ju Ma, "A strategy for determining a jacobi specific dataflow processor," in *Proceedings of* 11^{th} *Int. Conference of Applications-specific Systems, Architectures and Processors*, Zurich, Switzerland, July 14-16 1997, pp. 53 – 64.

[2] Bart Kienhuis, Ed Deprettere, Kees Vissers and Pieter van der Wolf, "A quantitative approach to dataflow architecture design," in *Proceedings of the ProRISC/IEEE Workshop on Circuits, Systems and Signal Processing.* STW/ProRISC, Nov. 27 - 28 1996, pp. 189 – 194.

[3] Bart Kienhuis, Ed Deprettere, Kees Vissers, Pieter van der Wolf, "An approach for quantitative analysis of application-specific dataflow architectures," in *Proceedings of* 11^{th} *Int. Conference of Applications-specific Systems, Architectures and Processors*, Zurich, Switzerland, July 14-16 1997, pp. 338 – 349.

[4] J.H. Justice, N.L Owsley, J.L Jen, and A.C. Kak, *Array Signal Processing*, Prentice-Hall, 1985.

[5] Boaz Porat, *Digital Processing of Random Signals*, Prentice-Hall, 1993.

[6] G.H. Golob and C.F. Van Loan, *Matrix Computations*, John Hopkins University Press, second edition, 1989.

[7] Arjan J.C. van Gemund, "Performance modeling with pamela: An introduction," Tech. Rep., Delft Institute of Technology, 1992.

[8] M.Nijweide, "The pamela run-time library version 1.3: Extensions and applications," Tech. Rep., Delft Institute of Technology, 1995.

[9] A.O. Looye, "Multiport memory and floating point cordic pipeline in jacobium processing elements," in *Proceedings of the ProRISC/IEEE Workshop on Circuits, Systems and Signal Processing*, Nov. 27 – 28 1997.

[10] P. Bollini, L. Chisci, A. Farina, M. Giannelli, L. Timmoneri, and G. Zappa, "Qr vs. iqr algorithms for adaptive signal processing: performance evaluation for radar applications," in *IEE Proceedings - Radar, Sonar and Navigation*, October 1996, vol. 143, pp. 328–341.

[11] Edwin Rijpkema, Ed F. Deprettere, and Gerben Hekstra, "An approach for the mapping of jacobi algorithms onto a jacobi specific dataflow processor," in *Proceedings of the ProRISC/IEEE Workshop on Circuits, Systems and Signal Processing*, Nov. 27 – 28 1997.