High Level Modeling for Parallel Executions of Nested Loop Algorithms*

Ed F. Deprettere and Edwin Rijpkema
Leiden University
2333 CA Leiden, The Netherlands
{edd,rypkema}@liacs.nl

Paul Lieverse
Delft University of Technology
2628 CD Delft, The Netherlands
lieverse@cas.et.tudelft.nl

Bart Kienhuis
University of California
Berkeley, CA 94720, USA
kienhuis@eecs.berkeley.edu

Abstract

High level modeling and (quantitative) performance analysis of signal processing systems requires high level models for the applications (algorithms) and the implementations (architecture), a mapping of the former into the latter, and a simulator for fast execution of the whole. Signal processing algorithms are very often nested-loop algorithms with a high degree of inherent parallelism. This paper presents - for such applications - suitable application and implemetation models, a method to convert a given imperative executable specification to a specification in terms of the application model, a method to map this specification into an architecture specification in terms of the implementation model, and a method to analyze the performance through simulation. The methods and tools are illustrated by means of an example.

1 Introduction

A new kind of embedded architectures is emerging that is composed of a microprocessor, some memory, and a number of dedicated coprocessors that are linked together via some kind of programmable interconnect (See Figure 1). These architectures are devised to be used in real-time, high-performance signal processing applications. Examples of these new architectures are the *Prophid* architecture [1], The *Jacobium* architecture [2], and the *Pleiades* architecture [3], to be used in respectively, video consumer appliances, adaptive array signal processing, and wireless mobile communications . These architectures have in common that they exploit instruction level parallelism offered by the microprocessor and coarse-grained parallelism offered by the coprocessors. Given a set of applications, the hardware/software codesign problem is to determine what needs to execute on the microprocessor and what on the coprocessors. Furthermore, what should each coprocessor contain, while being programmable enough to support the set of applications.

The applications that need to execute on the architectures are typically specified using an imperative model of computation, most commonly C or Matlab. In Figure 1, for example, we show an algorithm written in Matlab. Although the imperative model of computation is well suited to specify applications, it does not reveal parallelism due to its inherent sequential nature. Compilers exist that are able to extract instruction level parallelism from the original specification at a very fine level of granularity. They are, however, unable to exploit coarse-grained parallelism offered by the coprocessors of the architecture. This makes the mapping of the applications onto the architecture difficult.

Instead, a better specification format would be to use an inherently parallel model of computation like *Process Networks* [4, 5]. This describes an application as a network of concurrently executing processes. It describes parallelism naturally from the very fine-grained to the very coarse-grained, it does not pre-impose any particular schedule, and it describes each process in a

*To be presented at the IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP'2000), July 10 – 12 2000, Boston Massachusetts, USA.

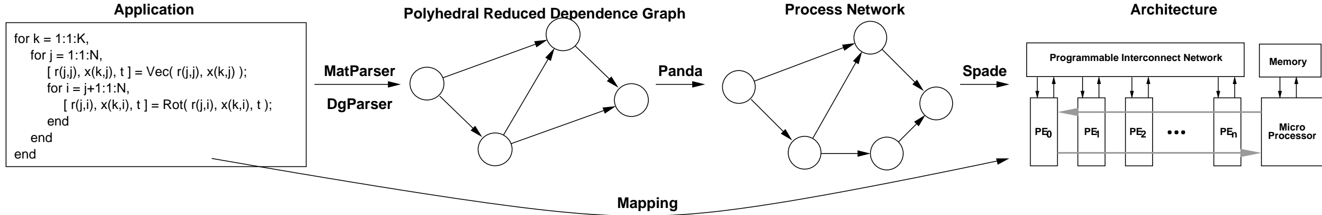


Figure 1. Mapping the application onto an architecture is difficult because the Model of Computation of the application does not match the way the architecture operates.

process network using an imperative language. The mapping then becomes putting the processes either on a microprocessor or on a coprocessor as shown by tools like ORAS [6], or SPADE [7]. Using these tools, a Y-chart [8] can be constructed, allowing the quality assessment of mappings on architectures.

This paper describes the *Compaan* tool that automatically transforms a Matlab application into a process network description, as shown in Figure 1. It converts a Matlab application into a *polyhedral reduced dependence graph*, that is subsequently converted into a *process network* description. The Compaan tool is confined to operate on affine nested loop programs (NLP) [9], but the applications of interest are often described this way.

The paper also describes the *Spade* tool that maps the *process network* description into an *architecture network* description and allows simulation of the resulting system thereby using some metrics collectors to analyze the performance of the system. From here, both the application and the architecture can be tuned (without violating the model rules) to explore the whole design space.

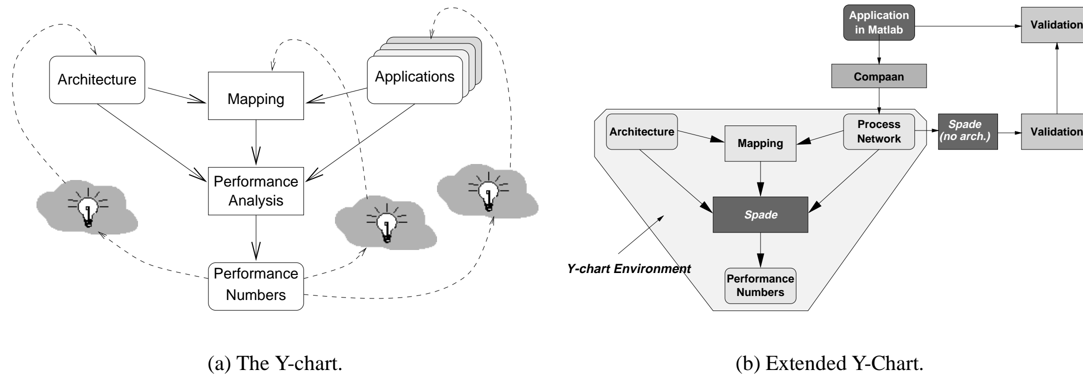
The outline of the paper is as follows. Section 2 gives the *Y-chart* exploration scheme underlying our concept of modeling and simulation. in Section 3 we present the Matlab-to-Process Network compiler *Compaan*. Section 4 briefly reviews the tools *MatParser* and *DgParser* that generate from a given imperative specification a single assignment code and a reduced dependence graph, respectively. Section 5 introduces the tool that generates the processes in the Process Network specification of an application. Section 6 describes the mapping and simulation tool *Spade* which is used in the Y-chart exploration scheme. Results are given in Section 7.

2 The Modeling Concept

In line with Kienhuis et al. [10] we advocate that the development of heterogeneous architectures should follow a general scheme, which can be visualized by the Y-shaped figure in Figure 2(a), the *Y-chart*. In the upper right part of Figure 2(a) is the set of applications that drives the design of the *architecture*. Typically, a designer studies this set of applications, makes some initial calculations, and proposes an architecture. The effectiveness of this architecture is then to be evaluated and a comparison with alternative architectures is to be made. Architectures are evaluated quantitatively by means of *performance analysis*. For this performance analysis, each application is *mapped* onto the architecture and the performance of each application–architecture–mapping combination is evaluated. The resulting performance numbers may inspire the architecture designer to improve the architecture. The designer may also decide to restructure the application(s) or to modify the mapping of the application(s). These designer actions are denoted by the light bulbs in Figure 2(a).

In this approach, it is assumed that the applications and the architecture are specified in terms of a model of computation and a model of implementation, respectively. For the parallel execution of signal processing nested loop programs (NLP), a natural model of computation is the *Process Network* model [4, 5] which is quite different from the usual imperative model of computation in which the applications are commonly specified. It is, therefore, necessary to provide a compiler that extracts the available parallelism from an application described as an NLP, say in Matlab, and automatically convert it into a process network specification. We thus extend the Y-chart environment shown in Figure 2(a) to the environment shown in Figure 2(b).

The specification of the architecture is in terms of a model of implementation, which is the interconnection of *generic building blocks* that are representative for the different types of resources in an architecture. These resources are for example processing resources, communication resources, and memory resources. The mapping of an application into an architecture is by means of *symbolic instruction traces* that connect the application model to the implementation model. This way, the functional behavior of the application resides in the model of computation and the timing behavior is confined to the model of



(a) The Y-chart.

(b) Extended Y-Chart.

Figure 2. The Y-chart (a) and the extended Y-chart (b).

implementation. Thus, compared to the laborious work of writing fully functional architecture models this can save a designer a lot of time, and therefore enables the exploration of alternative architectures.

In the next section, we focus on the upper right corner in Figure 2(b) which is implemeted in our *Compaan* tool. In Section 6, we take a closer look at the Y-chart part in Figure 2(b) which is implemented in our *Spade* tool.

3 The Compaan tool

We developed the Compilation of Matlab to Process Networks (Compaan) tool, which transforms a nested loop program written in Matlab into a process network specification. The tool does this transformation in a number of steps, shown in Figure 3, leveraging a lot of techniques available in the Systolic Array community [11]. In Figure 3, a box represents a result and an ellipsoid represents an action or tool.

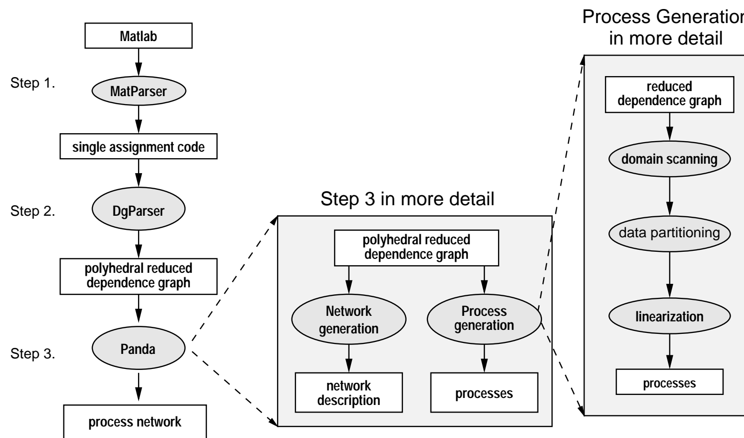


Figure 3. Compaan consists of three tools that transform a Matlab specification into a process network specification.

Compaan starts the transformation by converting a Matlab specification into a *single-assignment code* (SAC) specification. This describes all parallelism available in the original Matlab specification. Next, it derives the *polyhedral reduced dependence graph* (PRDG) specification from the SAC. From this PRDG, the network description and the individual processes are derived. The three steps done in Compaan are realized by separate tools, respectively, *MatParser*, *DgParser*, and *Panda*.

The last mentioned tool, Panda, uses the PRDG description to generate the network description and the contents of the

processes in the process network description. The generation of the processes is further decomposed into *domain scanning*, *data partitioning*, and *linearization*. We elaborate on these tools in Section 4 and Section 5.

In the Matlab–to–Process Network compilation, the role of the PRDG is crucial. A PRDG is a compact representation of an NLP's dependence graph. It is a directed graph $G = (V, E)$, where V is a set of *node domains* and E is a set of *edge domains*. The PRDG representation of the algorithm in Figure 1 is shown in Figure 4.

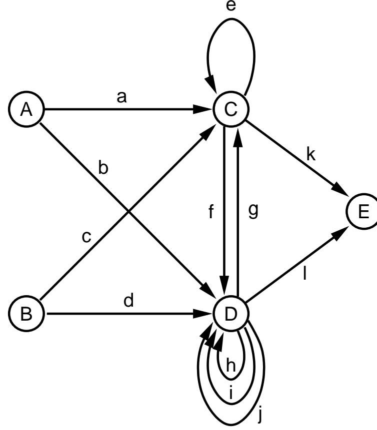


Figure 4. An example of a polyhedral reduced dependence graph.

3.1 Node domain

A node domain is characterized by 1) an iteration domain $\mathcal{I} = \{\mathbf{i} = L\mathbf{k}+\mathbf{m} \wedge \mathbf{k} \in \mathcal{P} \cap \mathbb{Z}^n\}$ where $\mathcal{P} = \{\mathbf{x} \in \mathbb{Q}^n \mid A\mathbf{x} \leq \mathbf{b}\}$ is a polytope, 2) a function, and 3) a set of port domains. Here, L and A are integral matrices, and $\mathbf{i}$, $\mathbf{k}$, $\mathbf{m}$, and $\mathbf{b}$ are integral vectors. The function resides in each point in the iteration domain. The function takes its arguments from its *input ports* and returns values to its *output ports*. A particular input port or output port belongs to the node domain's *input port domain* (IPD) and *output port domain* (OPD), respectively.

3.2 Edge domain

An edge domain is an ordered pair (v_i, v_j) of node domains together with an ordered pair (p_i, p_j) of port domains, where p_i is the OPD of v_i and p_j is the IPD of v_j. p_i is implicitly defined by an affine mapping M from IPD to OPD which expresses the data dependency between the output port of the function in v_i and the input port of the function in v_j.

3.3 Example

To illustrate the notion of node and port domains, we show in Figure 5 part of node domain C in Figure 4. The figure shows the node domain (a), its iteration domain with the iterators i and j -(we have left out the iterator k)- (b), its port domains (c)-(f), and its view as it appears in the PRDG (g). Thus, the four port domains (c)-(f) partition the node domain (a).

In (c) and (d), we show IPDs and in (e) and (f), we show OPDs. In (c) we identify two IPDs, ipd_1 and ipd_2. In (e) we identify two OPDs, opd_1 and opd_2. The figure shows one edge domain between opd_1 of port domain (e) and ipd_1 of port domain (c).

The PRDG is the intermediate specification between the given Matlab specification and the required process network specification.

4 MatParser & DgParser

In the path from Matlab to the PRDG, Compaan uses the tools *MatParser* [12, 9] and *DgParser* [9]. MatParser is an *array dataflow analysis* compiler that finds all parallelism available in NLPs written in Matlab using a very aggressive data -

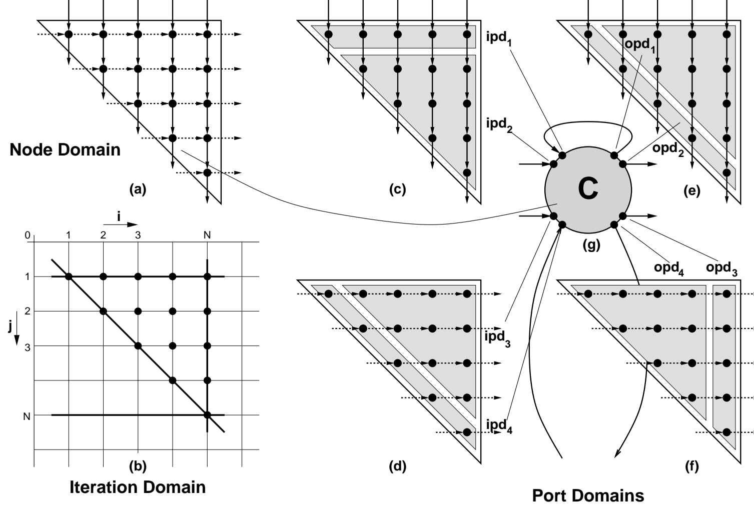


Figure 5. A node domain and its corresponding port domains.

dependency analysis technique based on integer linear programming [13]. We focus on Matlab since many signal-processing algorithms are written in this language. Just by writing another language front-end, MatParser can also operate on NLPs written in other languages, for example C.

MatParser finds whether two variables are dependent on each other, and moreover, at which iteration. It partitions the iteration space defined by the for-next loops, and gives the dependence vector between partitions. For the simple program given in Figure 1, MatParser solves about a hundred parametric integer program problems to find all data-dependencies.

In Figure 6, part of the output of MatParser is shown for the algorithm given in Figure 1. It shows how the iteration space spanned by the for-next iterators k and j is partitioned using if/else statements. Consequently, for different partitions, different data-dependencies may apply. In case of input argument in_0 of function `Vec`, either a value previously defined by function `Vec` should be used (i.e., $r_1(k-1, j)$, defining the data-dependency $M()$, or a value from the original r-matrix (i.e., $r(j, j)$) should be used.

DgParser converts the SAC description into the PRDG description, which is a straightforward conversion. Accordingly, the shape of the node domain is given by the way the for-next loops are defined and the partitioning of the node-domain corresponds with the if/else conditions. The terms `ipd` and `opd` used in Figure 6 relate to the IPD and OPD defined in Section 5.

5 Panda

Once DgParser has established a PRDG model of an algorithm, the Panda tool can generate a network description and the individual processes. The network description is straightforward, as it follows the topology of the PRDG. Each node in the PRDG is mapped onto a single process and each edge is mapped onto an unbounded FIFO. In case of Figure 4, nodes A, B, C, D, and E will define a process and edges a to l will define an unbounded FIFO.

As shown in Figure 3, the Panda tool divides the generation of a process into three different steps: domain scanning, data partitioning, and linearization, which we now discuss in more detail using a running example shown in Figure 7.

This figure shows two process network nodes, called the producer and the consumer and an unbounded FIFO channel for the communication between the two. Also shown is part of a PRDG, consisting of node domains ND_p and ND_c, the OPD of ND_p (shaded triangle), the IPD of ND_c (shaded triangle), and the affine mapping M from IPD to OPD. Panda generates the top-part of the figure from the bottom-part.

5.1 Domain scanning

The first step in Panda is to scan the node domains by lexicographically ordering the points in the node's iteration domain.

```
%% Single Assignment Code Generated by MatParser
for k = 1 : 1 : K,
   for j = 1 : 1 : N,

      if k-2>= 0,
        [ in_0 ] = ipd( r_1( k-1, j ) );
      else %% if -k+1 >= 0
        [ in_0 ] = ipd( r( j, j ) );
      end
      if j-2>= 0,
         [ in_1 ] = ipd( x_1( k, j-1, j ) );
      else %% if -j+1 >= 0
         [ in_1 ] = ipd( x( k, j ) );
      end

      [ out_0, out_1, out_2 ] = Vec( in_0, in_1 );

      [ r_1( k, j) ]  = opd( out_0 );
      [ x_1( k, j) ]  = opd( out_1 );
      [ t_1( k, j) ]  = opd( out_2 );
   end
end
```

Figure 6. Single Assignment Code

Thus, given the iteraton domains $\mathcal{J} = \{\mathbf{j} = L_p\mathbf{k} + \mathbf{m}_p \wedge \mathbf{k} \in \mathcal{P}_p \cap \mathbb{Z}^2\}$ for ND_p and $\mathcal{I} = \{\mathbf{i} = L_c\mathbf{k} + \mathbf{m}_c \wedge \mathbf{k} \in \mathcal{P}_c \cap \mathbb{Z}^2\}$ for ND_c, respectively, and the iterator scan orders, say (j_2, j_1) for ND_p and (i_1, i_2) for ND_c, the task is to return a lexicographical ordering of the domains in terms of a nested loop.

The Fourier-Motzkin procedure [14] is used to accomplish this. Fourier-Motzkin (FM) finds the bounderies of the iterators, given the iterator scan orders. Thus, FM returns for `sort order`(j_2, j_1): $\{0 \leq j_2 \leq N\ ,\ j_2 \leq j_1 \leq N\}$ and for `sort order`$(i_1, 1_2)$: $\{0 \leq i_1 \leq N\ ,\ 0 \leq i_2 \leq i_1\}$. The conversion to a nested loop scanning is then straightforward.

5.2 Data partitioning

MatParser generates a SAC description in which only the IPDs are explicitly specified. This means that the input arguments in_0 and in_1 in Figure 6, are surrounded by if/else statements, while the output values out_0, out_1, and out_2 are not. A consequence of this is that output values can be generated that are never used by some input domain, or that one OPD produces

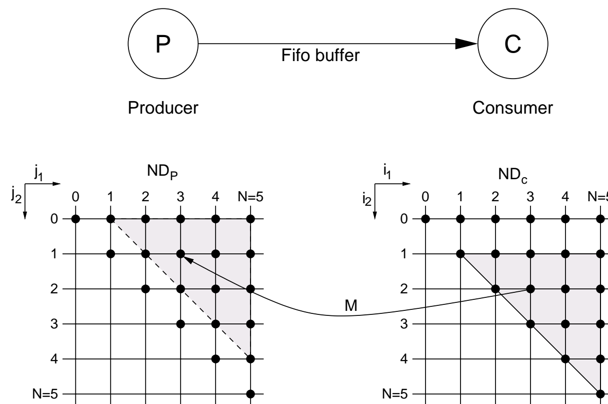


Figure 7. Mapping of a PRDG onto a Process Network, running example.

tokens for more than one IPD. The second step in Panda is to make the OPDs explicit.

Making the output port domains explicit is illustrated in Figure 8. It shows two communicating node domains ND_p and ND_c. The tokens produced by port domain P_p of node domain ND_p are to be consumed by port domain P_c of node domain ND_c, as described by the data-dependency with mapping M. Port domain P_p is an OPD and port domain P_c is an IPD. To make P_p explicit, Panda applies $M()$ that is derived by MatParser, to IPD P_p, which is an operation on $\mathbb{Z}$-polyhedra [15].

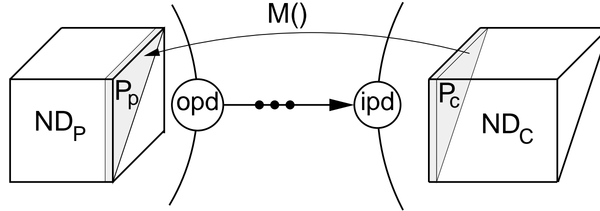


Figure 8. Making the output port domain explicit.

The procedure goes as follows. Starting point is the relation between IPD and OPD through M. Given the IPD (MatPars output) $\{0 \le i_1 \le N \ , \ 0 \le i_2 \le i_1\}$ of ND_c, and the affine mapping M: $\{j_1 = i_1 \ , \ j_2 = i_2 - 1\}$, the OPD $\{0 \le j_2 \le N-1 \ , \ j_2+1 \le j_1 \le N\}$ of ND_p is found. Comparing these port domains with the respective node domains, it follows that what remains to be checked at run time - to detect the port domains while scanning the node domains - is whether $j_2 + 1 \le j_1$ in ND_p and $1 \le i_2$ in ND_c.

5.3 Linearization

The channels between processes are one-dimensional FIFO buffers. Therefore, the order in which a consuming process reads tokens from a channel must be the same as the order in which tokens are written onto the channel by the producing process. Of course, the consuming process will in general use the read tokens in a different order (out-of-order consumption). The channel's FIFO and the consumer process's reorder memory are modeled as one one-dimensional memory m. This is shown in Figure 9.

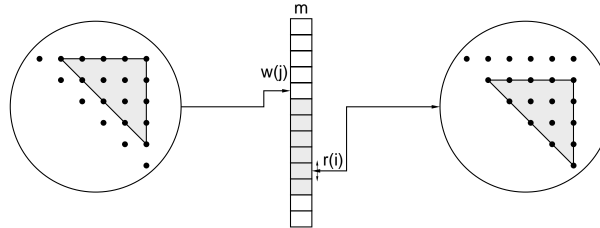


Figure 9. The producer-to-consumer chaneel FIFO and the consumer's memory modeled as a linar memory m.

m is a logical storage structure (LSS) [16], and there is one LSS for each opd/ipd pair. The producer node writes tokens into the LSS in the order given by a *write polynomial* $w(\mathbf{j})$. The consumer node consumes tokens from this LSS in the order given by a *read polynomial* $r(\mathbf{i})$. The write polynomial follows from a ranking of the OPD of the producer. The ranking follows the order resulting from the domain scanning and data partitioning steps. The read polynomial satisfies $r(\mathbf{i}) = w(M(\mathbf{i}))$, where $M(\mathbf{i})$ is the affine mapping from the IPD of ND_c to the OPD of ND_p. The ranking in ND_p in the running example is shown in Figure 10. The corresponding write polynomial is given as

$$w(j_1, j_2) = -\frac{1}{2}j_2^2 + (N - \frac{1}{2})j_2 + j_1 - 1 \tag{1}$$

The corresponding read polynomial for ND_c is given by

$$\begin{aligned} r(i_i, i_2) &= w(M(i_1, i_2)) = w(i_1, i_2 - 1) \\ &= -\frac{1}{2}i_2^2 + (N + \frac{1}{2})i_2 + i_1 - (N + 1) \end{aligned} \tag{2}$$

The linearization method in Panda relies on methods to count the number of integral points contained within a polytope using so-called *Ehrhart Polynomials* [17]. These methods are implemented in the library *PolyLib* [18].

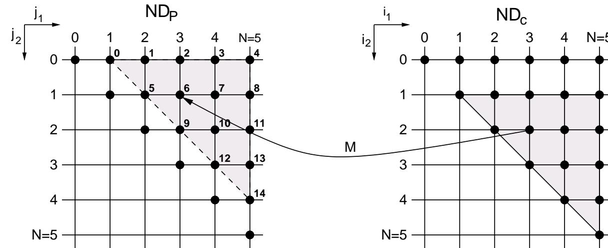


Figure 10. Ranking of node domains to derive read and write polynomials for addressing the LSS m between producer and consumer.

The three steps *domain scanning*, *data partitioning*, and *linearization* result in a control program as shown in Figure 11 for the running example.

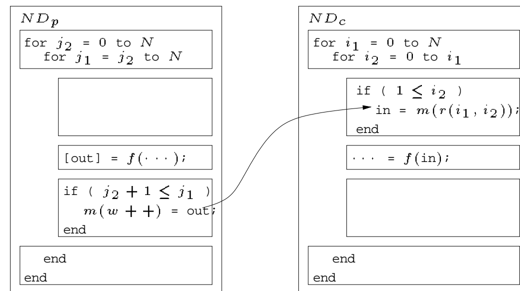


Figure 11. Control program for writing in-order to the channel and reading - possibly out-of-order - from memory

.

As can be seen from the figure, ND_p's **j**-loopnest scans the port domain and produces in each and every point the output of the function's firing in these points. If the condition $j_2 + 1 \leq j_1$ is true, then the output values are put in-order to the channel. ND_c gets these values from the channel and puts them into a reordering memory. ND_c's **i**-loopnest scans the port domain and if the condition $1 \leq i_2$ is satisfied, then the input value for the function residing in the node domain at the scan point is read from the reordering memory at address $m(r(\mathbf{i})) = m(w(M(\mathbf{i})))$.

6 The Spade tool

The Spade tool was designed to execute Y-charts as shown in Figure 2(a). Spade makes a clear distinction between applications and implementations. An application imposes a *workload* on the *resources* provided by an architecture. The workload

consists of both the *computation workload* and the *communication workload*. The resources can be *processing resources*, such as programmable cores or dedicated hardware units, *communication resources*, such as bus structures, and *memory resources*, such as RAMs or FIFO buffers. The architecture design process is concerned with the specification of the resources that can best handle the workloads imposed by the target applications.

In Spade, applications and architectures are modeled separately. An application is modeled as a Kahn Process Network (KPN) that is a functional model relatively free of architectural aspects. Vice versa, an architecture is modeled as an implementation network in which the resources are taken from a library of generic building blocks. The architecture is constructed for all applications from the benchmark set. Such a *decoupling* enables *reuse* of both applications and architectures and facilitates an explorative design process in which applications are subsequently mapped onto an architecture.

In order to evaluate the performance of an application–architecture–mapping combination, we must provide the interfacing of application models to architecture models, including specification of mappings. For this purpose, we extend a technique called *trace-driven simulation*. This is a simulation technique that has been applied extensively for memory system simulation in the field of general-purpose processor design [19]. We use the technique for performance analysis of heterogeneous systems. Each process in the KPN produces upon execution a so-called *trace*, which represents the workload imposed on an architecture by that process. Thus, a trace contains information on the communication and computation operations performed by an application process. These operations may be *coarse-grain* as opposed to classical trace-driven simulation, in which traces contain information on fine-grain RISC operations only. The traces drive computation and communication activities in the architecture. These activities are executed as specified by the architecture model. During this execution, *time* gets assigned to all events that occur in the architecture model and the performance of the execution of the application on the architecture can be measured. An example of an application specified as a KPN is shown in Figure 12.

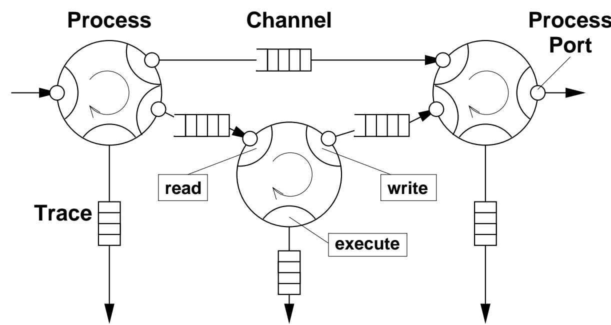


Figure 12. Example of an application model. Processes are depicted as circles, small circles are process ports, and the circle segments in the processes are API functions.

Spade offers an Application Programmers Interface (API) for application modeling that contains the following three functions.

- A *Read* function. This function is used to read data from a channel via a process port. Furthermore, the function generates a *trace entry* - a symbolic read instruction - in the trace of the process by which it is invoked, reporting on the execution of a read operation at the application level.
- A *Write* function. This function is used to write data to a channel via a process port. It also generates a trace entry - a symbolic write instruction, reporting on the execution of a write operation.
- An *execute* function. This function performs no data processing, but only generates a trace entry, reporting on processing activities at the application level. The Execute function takes a *symbolic instruction* as an argument in order to distinguish between different processing activities. For example, such an instruction may correspond to a **Vec** or **Rot** function in the application code in Figure 1.

The trace entries generated by the read and write functions represent the *communication workload* of a process. The trace entries generated by the Execute function represent the *computation workload* of a process. The trace entries can be used either to drive the architecture simulation, or, when executing the application *stand-alone*, to analyze the computation and communication workload of an application. Such an analysis may return the number of times an operation in a process fires

and how many tokens were transported over a FIFO channel between processes. This is shown in Figure 13, which describes the same network as given in Figure 4. The values of the parameters N and K - see Figure 1 - have been 6 and 100 respectively.

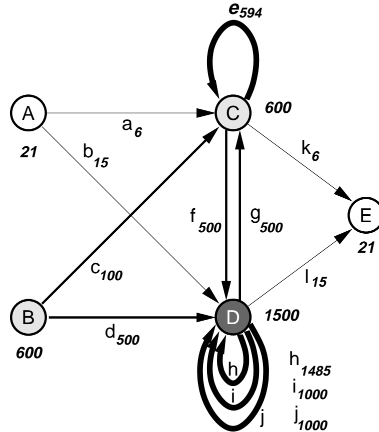


Figure 13. The PRDG with the firing numbers per process and channel/

Thus, processes A and E have 21 executes, processes B and C 600 and so on. In the figure, processes that execute more frequently are colored darker and the channels have a width depending on their communication load. We can see that, for example, edge b transported 15 tokens, while edge g transported 500 tokens and edge e transported 594 tokens.

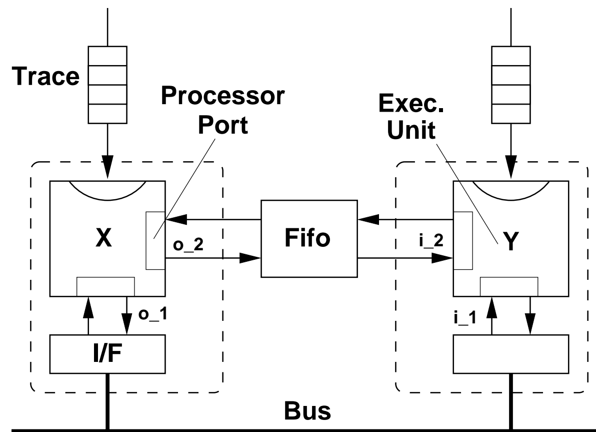


Figure 14. Example of an architecture model. There are two processing resources (the dashed boxes), that each are composed of a Trace Drive Execution Unit and an interface, one FIFO, and a bus.

An example of an architecture model is shown in Figure 14. The processing resources in the architecture model take the traces generated by the application as input. We have taken a modular approach to allow the construction of a great variety of processing resources from a small number of basic building blocks. A processing resource is built from the following two types of blocks.

- A *trace driven execution unit (TDEU)* which interprets trace entries. The entries are interpreted in the order in which they are put in the trace, thereby retaining the order of execution of the application process. A TDEU has a configurable number of I/O ports. Communication via these I/O ports is based on a generic protocol.
- A number of *interfaces*, which connect the I/O ports of a TDEU to a specific communication resource. An interface translates the generic protocol into a communication resource specific protocol, and may include buffers to model input/output buffering of processing resources. Currently, we have interfaces for communication via a direct link, via a shared bus, and via shared memory, both buffered and unbuffered.

Apart from the TDEU and interface blocks, the current library contains some generic bus blocks, including bus arbiters, and a generic memory block. All blocks are parameterized. For each instantiated TDEU a list of *symbolic instructions* and their *latencies* and *throughputs* have to be given. This list specifies which instructions from the traces can be executed by the processing resource and how many cycles each instruction takes when executed on this processing resource. Both latencies and throughputs can be obtained from a lower level model of a processing resource, from estimation tools or they can be estimated by an experienced designer. For instances of the interface blocks, buffer sizes can be given. For a bus instance, the bus width, setup delay, and transfer delay can be specified. Recall that the generic building blocks are abstract performance models; they only model the timing and synchronization of the architecture. The application model captures the functional behavior.

Once both an application model and an architecture model have been defined, mapping can be performed. This means that the workload of the application has to be assigned to resources in the architecture as follows.

- Each process is mapped onto a TDEU. This mapping can be many-to-one, in which case the trace entries of the processes need to be scheduled by the TDEU. The scheduling policy can be selected and specified by the user.
- Each process port is mapped one-to-one onto an I/O port. This mapping also implicitly maps the channels onto a combination of communication resources and memory resources, possibly including user defined blocks, such as a specific bus model or memory interface. Typically, these resources do not have an equivalent element in the application model.

The simulation of the application model is based on the Pamela [20] multi-threading environment, where each Kahn process is executed in a separate thread. The simulation of the architecture model is currently based on TSS (Tool for System Simulation), which is a Philips in-house architecture modeling and simulation framework [21]. The library of generic blocks is implemented as a library of TSS modules. The generic building blocks already contain collectors for different performance metrics. The metrics for which data is collected during simulation is resource specific. More on Spade can be found in [22].

7 Modeling and simulation results

For the application shown in Figure 1 we have derived the PRDG and KPN specifications shown in Figure 4. From the execution and transmission counts in Figure 13, we see that some processes execute many times (i.e., node B, C, and D), while others do so sporadically (i.e., A and E). Based on this insight, we suggested a partition for the architecture as shown in Figure 1; the frequently executing processes are mapped on coprocessors whereas the incidentally executing processes are put on the microprocessor. Consequently, channels $\{a, b, i, k\}$ map onto the low-bandwidth communication structure that connects the coprocessors with the microprocessor. Edges $\{c, d, f, g\}$ map onto the programmable interconnect network, which is the high - bandwidth communication structure. Edge e and edges $\{i, j, h\}$ map onto internal communication structures inside the coprocessor for node C and D, respectively. This very high-bandwidth communication is thus kept local to the coprocessors. The specific architecture specified and used in Spade is shown in Figure 15.

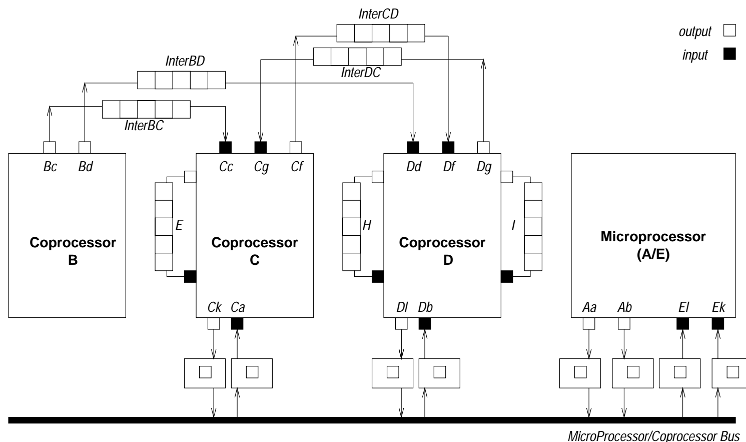


Figure 15. The architecture for the modeling and simulation example

The architecture consists of a microprocessor (Mp), three coprocessors (Cp), a Mp-to-Cp bus with businterfaces with a single buffer space, and FIFO interconnects local to the coprocessors. A bus-transfer takes d cycles and a FIFO-channel transfer takes one cycle. Coprocessor C and coprocessor D receive symbolic *execute* instructions with two parameters, *latency* (l) and *throughput* (t). Simulation results are shown in Tablee 1 an Table 2. The values of the parameters l and t are 21 and 1 cycles, respectively.

Table 1. Simulated execution times for problem sizes $N = 4, 6, 12, 16$ and $K = 100, 200$ and bus-delay $d = 0, 10$.

	$K = 100$		$K = 200$	
N	d=0	d=10	d=0	d=10
4	2153	2256	4253	4356
6	2204	2467	4304	4567
12	6706	7908	13306	14508
16	12102	14316	24102	26316

Table 2. Problem period α in cycles for problem sizes $N = 4, 6, 12, 16$ and bus-delay $d = 0, 10$.

N	$d = 0$	$d = 10$
4	21	21
6	21	21
12	66	66
16	120	120

In Table 2, the problem period α is defined as $\alpha = \frac{T(K+\Delta K)-T(K)}{\Delta K}$ in cycles. For values of N for which $\frac{1}{2} \times (N-1) \times N \leq l$, the *execute* instruction's latency is dominating.

8 Conclusion

With the toolset presented in this paper, certain nested loop Matlab programs can be converted to Kahn Process Networks and mapped onto a high-level architecture description for simulation and performance analysis on a high level of abstraction. Not discussed in the paper is a tool for performing transformations on the intermediate PRDG specification of the application. Examples of such transformations are *index transformations* and PRDG *unrolling*. In the example given in the paper, the PRDG was index transformed to improve the utilization of the pipelined operations **Vec** and **Rot** in the architecture. The transformation *unrolling* typically results in a higher degree of parallelism. All elements of the Compaan tool are implemented in Java. Part of it is integrated in the Ptolemy II framework (see `www.gigascale.org/compaan`).

9 Acknowledgement

The authors want to thank Pieter van der Wolf and Kees Vissers from the Philips Research Laboratories for valuable discussions and support. They are also indebted to Vincent Loechner from INRIA for the Polylib guidance and support. The work was supported in part by the universities of Delft and Leiden, in part by Philips Research and in part by the MARCO/DARPA Gigascale Silicon Research Center.

References

[1] Jeroen A.J. Leijten, Jef L. van Meerbergen, Adwin H. Timmer, and Jochen A.G. Jess, "Prohid, a data-driven multiprocessor architecture for high-performance dsp," in *Proc. ED&TC*, Mar. 17-20 1997.

[2] Edwin Rijpkema, Ed F. Deprettere, and Gerben Hekstra, "A strategy for determining a jacobi specific dataflow processor," in *Proceedings ASAP'97 conference*, July 1997.

[3] Arthur Abnous and Jan Rabaey, "Ultra-low-power domain-specific multimedia processors," in *VLSI Signal Processing, IX*, 1996, pp. 461–470.

[4] Gilles Kahn, "The semantics of a simple language for parallel programming," in *Proc. of the IFIP Congress 74*. 1974, North-Holland Publishing Co.

[5] Edward A. Lee and Thomas M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–799, May 1995.

[6] Bart Kienhuis, Ed Deprettere, Kees Vissers, and Pieter van der Wolf, "The construction of a retargetable simulator for an architecture template," in *Proceedings of 6th Int. Workshop on Hardware/Software Codesign*, Seattle, Washington, Mar. 15–18 1998.

[7] Paul Lieverse, Pieter van der Wolf, Ed Deprettere, and Kees Vissers, "A methodology for architecture exploration of heterogeneous signal processing systems," in *Proceedings of the 1999 IEEE Workshop in Signal Processing Systems*, Taipei, Taiwan, 1999.

[8] Bart Kienhuis, *Design Space Exploration of Stream-based Dataflow Architectures: Methods and Tools*, Ph.D. thesis, Delft University of Technology, Jan. 1999.

[9] Peter Held, *Functional Design of Data-Flow Networks*, Ph.D. thesis, Dept. EE, Delft University of Technology, May 1996.

[10] Bart Kienhuis, Ed Deprettere, Kees Vissers, and Pieter van der Wolf, "An approach for quantitative analysis of application-specific dataflow architectures," in *Proc. ASAP'97*, July 14-16 1997.

[11] S.Y. Kung, *VLSI Array Processors*, Prentice Hall Information and System Sciences Series, 1988.

[12] Bart Kienhuis, "Matparser: An array dataflow analysis compiler," Tech. Rep. UCB/ERL M00/9, University of California, Berkeley, CA-94720, USA, Feb. 2000.

[13] Paul Feautrier, "Parametric integer programming," *Recherche Opérationelle; Operations Research*, vol. 22, no. 3, pp. 243–268, 1988.

[14] C. Ancourt and F. Irigoin, "Scanning polyhedra with DO loops," in *Proc. ACM SIGPLAN '91*, june 1991, pp. 39–50.

[15] Patrice Quinton, Sanjay Rajopadhye, and Tanguy Risset, "On Manipulating $\mathbb{Z}$-polyhedra," Tech. Rep., Institut de Recherche en Informatique et Systèmes Aléatoires, 1996.

[16] Alco Looye, Gerben Hekstra, and Ed Deprettere, "Multiport memory and floating point cordic pipeline in jacobium processing element," in *Proceedings SiPS 98 Design and Implementation*, October 1998, pp. 406–426.

[17] Phillipe Clauss and Vincent Loechner, "Parametric analysis of polyhedral iteration spaces," *Journal of VLSI Signal Processing*, vol. 19, pp. 179–194, July 1998.

[18] Doran K. Wilde, "A library for doing polyhedral operations," M.S. thesis, Oregon State University, Corvallis, Oregon, Dec 1993, Also published in IRISA technical report PI 785, Rennes, France; Dec, 1993.

[19] Richard A. Uhlig and Trevor N. Mudge, "Trace-driven memory simulation: A survey," *ACM Computing Surveys*, vol. 29, no. 2, pp. 128–170, June 1997.

[20] Arjan J. C. van Gemund, "Performance prediction of parallel processing systems: The PAMELA methodology," in *Proc. 7th ACM Int. Conference on Supercomputing*, Tokyo, July 1993, pp. 318–327.

[21] Wido Kruijtzer, "TSS: Tool for System Simulation," *IST Newsletter*, vol. 17, pp. 5–7, Mar. 1997, Philips Internal Publication.

[22] Paul Lieverse, Pieter van der Wolf, Ed Deprettere, and Kees Vissers, "A methodology for architecture exploration of heterogeneous signal processing systems," *to appear in Journal of VLSI Signal Processing*, 2000.