

Deriving efficient control in Process Networks with Compaan/Laura

Steven Derrien, Alexandru Turjan, Claudiu Zissulescu, Bart Kienhuis, Ed Deprettere

Leiden Institute of Advanced Computer Science, Leiden, The Netherlands

Abstract:

At Leiden Embedded Research Center (LERC), we are building a tool chain called *Compaan/Laura* that allows us to map rapidly and efficiently signal processing applications written in Matlab onto reconfigurable platforms. In this chain, first the Matlab code is converted automatically to an executable Kahn Process Network (KPN) specification [6], then a tool called *Laura* transforms the PN specification into a design implementation described as synthesizable VHDL.

The applications targeted by Compaan are usually data-flow intensive, requiring large computational power. Therefore, an important issue in Laura is the derivation of efficient and scalable hardware control structures. This control is based on an abstract representation given as parametrized polytopes. Although this representation can be directly translated into nested guarded for-loops (very suitable for software implementation) its mapping to hardware is much more difficult. In this paper we investigate the opportunities of deriving different hardware realizations for this control, and explore the trade off between speed and resource usage.

Keywords: Parameterized Control; Hardware synthesis; Loop parallelization, Kahn Process Networks

Reference to this paper should be made as follows: Derrien, S., Turjan, A., Zissulescu, C., Kienhuis, B., and Deprettere, E. (xxxx) ‘Deriving efficient control in Process Networks with Compaan/Laura’, *International Journal on Embedded Systems*, Vol. x, No. x, pp.xxx–xxx.

1 INTRODUCTION

The aim of the *Compaan* framework [9, 17, 15] is to automatically transform applications in the field of signal and image processing to Process Networks (PNs). A PN is a deterministic model of computation that expresses an application in terms of distributed memory and distributed control. This makes a PN suitable for mapping on parallel architectures [17]. By writing the application in a subset of Matlab, the Compaan framework converts it into a PN representation. From this representation, using the Laura tool [21], it is possible to generate a hardware description of the KPN in VHDL which can be mapped on an FPGA based hardware platform.

During this conversion, each process is mapped to a hardware *virtual process* consisting of a Read, an Execute, and a Write unit that use a local hardware controller. This controller executes a particular control program that is derived by Compaan. This control program is referred to as *control* in this paper. Because of the inherent complexity of this control, deriving an efficient controller implementation (both in terms of area and speed) is mandatory to obtain good performance. In this paper, we investigate different strategies for deriving hardware realizations for this control and explore the trade off between

speed and resources usage. The rest of the paper is organized as follows : we first present the Compaan/Laura flow in order to understand how the hardware mapping is done by Laura. We then address the problem of the hardware controller generation and propose three distinct approaches: A simple control mechanism based on ROM tables, a parameterized controller, and a more sophisticated approach of controller called variant-controller. We will finish this paper by a discussion over the merits and the relevance of these approaches in the context of Compaan/Laura.

1.1 Compaan/Laura design flow

As mentioned in the introduction, we assume that the process networks are derived using the Compaan toolset. Compaan takes as input parameterized static nested loop programs written in Matlab.

The Compaan framework consists of four tools. The first tool, i.e. MatParser [8, 5], transforms the initial Matlab code into single assignment code (SAC) using exact data flow analysis [4, 13]. DgParser [15] converts the SAC into a Polyhedral Reduced Dependence Graph (PRDG) data structure, which is a compact mathematical representation of the Dependence Graph

(DG) in terms of polyhedra. Panda [18] then converts the PRDG into a PN, associating a process with each node of the PRDG.

The last tool of the flow, called *Laura*, is used to generate a VHDL description of an architecture from a PN description. During this step, each process of the PN is mapped to an abstract architectural model called *Virtual Processor*, which is represented in Figure 1. Each virtual processor is made of four

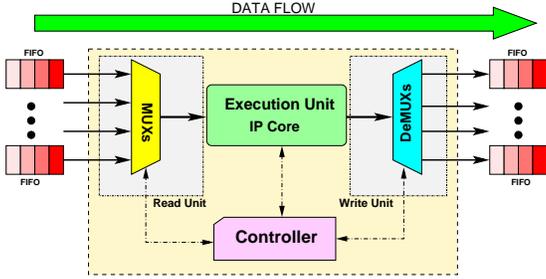


Figure 1: Hardware realization of a process

distinct components:

- An *Execute unit* that is the computational part of the virtual processor. This unit can be seen as a wrapper for an IP core which implements the functionality of the node associated to the process. Its interface consists of a number of *Input arguments* and *Output arguments*.
- A *Read unit* is responsible for assigning all input arguments with a valid token to the Execute Unit. We call an *Input Port* the input interface that connects the virtual processor with a FIFO channel. Since there are more input ports than arguments, the Read unit has to select from which port to read a token using the information provided by the controller unit.
- A *Write unit* is responsible for distributing the results of the Execute unit to the process output FIFOs. A write operation can be executed only when all the output arguments of the execute unit are available for the write unit. We call an *Output Port* the output interface that connects the virtual processor with a FIFO channel. Similarly to the Read unit, several output port may share the same execute unit output argument and the Write Unit has to select which port is to receive the token as provided by the Controller Unit.
- A *Controller unit* is responsible to synchronize all the processor's units, to keep track of how many iterations have already been executed, and to generate the activation signals for the input ports and output ports of a virtual processor.

2 Control Generation

The most critical component within a *Virtual Processor* is its controller unit. Its hardware realization must be fast enough not to slow the rest of the processor (i.e., it needs to be faster than

the operation speed of the IP core used in the Execute Unit), and must use as few resources as possible. Deriving the best hardware realization for this controller is hence key to obtain efficient and practical implementations for the processors. To have a better understanding of the activation sequences generated by the control program stored in the Controller Unit, we show in Figure 2 a simple producer/consumer pair example and detail its operation.

In a PN derived by Compaan, each process executes an internal function following a local schedule (which corresponds to a loop nest). At each execution (also referred to as an *iteration*), this function reads/writes data from/to different FIFOs. We call the *Input Port Domain* (IPD) of a process, the iteration sub-domain (which is a union of polytopes) for which the process reads data coming the same FIFO. Similarly, we call the *Output Port Domain* (OPD) of a process, the iteration sub-domain (again a union of polytope) at which the process writes data to the same FIFO. Each FIFO uniquely relates an input port to an output port.

In Figure 2, we show a fragment of a PN in which processes A and B communicate data over fifo channel FIFO1. In both processes, nestings of for-loops indicate the schedule in which the functions **FuncA** and **FuncB** are being executed. Depending on an iteration, some particular conditions, expressed by linear conditions in the if-statements, will be true and the result of **FuncA** will be sent to either FIFO0 or FIFO2. Independently of Process A, Process B executes a particular schedule given by its nested-for loops. Similarly, depending on a particular sequence, a particular token is either read from FIFO0 or FIFO1 and is given as input argument to **FuncB**. Deriving the control

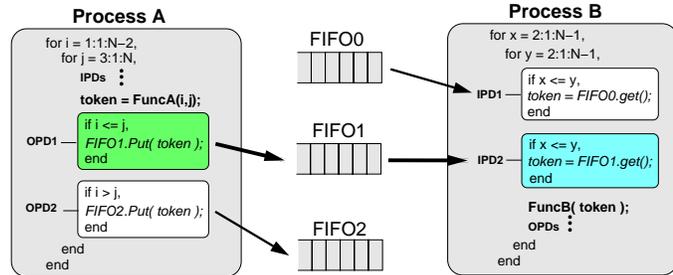


Figure 2: Two processes taken out from a Compaan generated PN

sequence for a virtual processor means that we need to know, for each process iteration, from which FIFOs data has to be read, and to which FIFOs data has to be written. Consequently, a tool like *Laura* needs to be able to generate, for each process, a hardware realization generating the corresponding control sequence. In this paper, we investigate three different approaches for this hardware derivation step:

- A ROM table based architecture.
- A (partitioned) parameterized predicate controller architecture.
- A run-time distance evaluation approach.

Because the control of a process as generated by Compaan is static, the sequence of reading/writing data from/to the FIFO channels is data-independent. Therefore, the simplest approach to implement in hardware the local schedule of a virtual processor is to use ROM memory tables to store the IPD/OPD activation code for the process domain iteration. The ROM table controller is the best solution to achieve small clock delay for the system, as no additional computation takes place in the Controller. However, this approach appears to be impractical for loop nest iterating over large domains, as the ROM size grows with the iteration space. Therefore these ROM tables can very quickly exceed the storage capacity of most FPGAs.

To overcome this issue, it is possible to apply compression algorithms on the content of these ROM and perform on-the-fly decompression of the ROM content. In the Compaan/Laura context, we must limit ourselves to very simple compression techniques, such as Run Length Encoding, since we want the decompression engine area overhead to be very limited and not to cause additional delays.

In Table 1, we show the ROM table sizes for a number of parameterized applications. The applications are: QR which is a matrix decomposition algorithm [19], Stereo vision which is a 1-D motion estimation algorithm [10], and Optical Flow which is a generic image restoration algorithm [11]. For each application, we give for a particular node in the network the size of the ROM table needed to program the Controller. The nodes shown correspond to the nodes with the biggest ROM table. For example, Node 4 of the PN representing Stereo vision, requires a controller ROM table of 459.016 bytes when selecting the parameter values $W = 320$ and $H = 200$. If we apply a simple run-length encoding scheme, we can compress this to 401.320 bytes. Suppose we map the node on a Virtex-II 6000 device. This device has a maximum of 3648K bits available. Hence, implementing the control table of a single processor would already consume 88% of the available memory bits on the FPGA, which is totally unacceptable.

	N	T	direct bytes	RLE bytes	% of mem FPGA
QR (node 4/5)	8	16	448	400	0.08
	16	64	7680	4160	0.9
	64	256	516096	78080	17.12
	W	H	simple	RLE	% of mem
Stereo vision (node 4/5)	320	200	459016	401320	88.00
	640	400	1941576	1698240	372.42
	1024	640	5072328	4437264	973.08
	W	H	simple	RLE	% of mem
Optical Flow (node 3/7)	320	200	944460	14850	3.25
	640	480	3808860	29850	6.54
	1024	764	9780540	47850	10.4

Table 1: Control ROM size for three different applications

Looking at Table 1, we conclude that for large domains, the ROM approach is non practical. For a small size of the QR-algorithm, however, the approach remains feasible and very fast.

Another way to realize the controller of the virtual processor is to evaluate the predicates as they appear in the original program of the processes of the PN. For example, in Figure 2, two processes are given. Process A has two OPDs that are guarded by two predicates $i \leq j$ and $i > j$. By evaluating these predicates at run-time, Process A can select the proper FIFO to send its result to. A big advantage of this approach is that the control is parameterized by the original parameters of the application. Because of the class of nested-loop program that Compaan accepts, all predicates represented by *linear expressions* depend only on the loop indices and parameters.

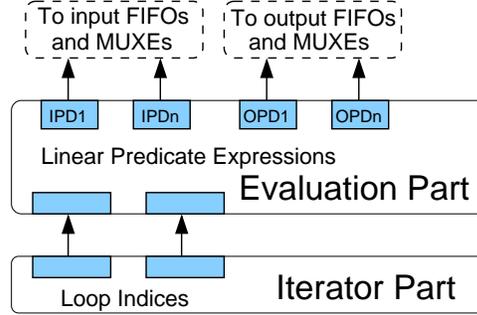


Figure 3: Structure of a Parameterized Predicated Controller

4.1 Hardware structure of the controller

As shown in Figure 3, a predicate controller consists of two parts : an iterator part and an evaluation part. In the *Iterator Part*, the Controller iterates over the loop domain and updates the loop indices according to the currently visited iteration. In the *Evaluation Part*, the Controller evaluates in parallel and at every iteration, all the *linear predicate expressions* associated to the activation of the IPDs and OPDs status. If we want to implement a predicate controller for Process A, the Iterator part provides the values of the iterators, i and j , that are used in the Evaluation part to evaluate the predicates $i \leq j$ and $i > j$. A parameterized predicated controller can be derived in four steps, as described below:

4.1.1 redundancy elimination

the first step consists in reducing the computational load by applying a common expression elimination to remove all redundant expressions present in the predicates associated to the IPDs/OPDs. This is a very effective step, as in the vast majority of the targeted applications, this optimization eliminates up to 80% of the computations involved in the controller.

4.1.2 dependence graph extraction

in the second step, a data-dependence graph is extracted. This dependence graph is associated to the linear expressions involved in the loop-iterators and IPDs/OPDs predicates : com-

putations from both the Iteration and Evaluation part are combined. During this step, we make the assumption that all the linear expressions appearing in the predicate have their terms sorted in increasing order of the depth of the loop-iterators. In other words, the most inner loop index should always appear as the last one evaluated in the *linear expression*. This is needed to ensure that the dependence graph gets sorted in the optimal way.

4.1.3 hardware mapping

in the third step, the data-dependence graph is mapped onto a specialized parallel data-path, by associating to each operation/node of the graph its hardware equivalent. We map the data-dependence graph as a pure combinational data-path with only its output being synchronized through registers. Additionally, a feed-back loop is created to enable the updating of the state of the loop iterators.

4.1.4 bitwidth selection

the last step of this datapath synthesis operation is to determine the bitwidth of the various operators involved in the data-path. This analysis is crucial since it allows to drastically improve both area usage and performance (a controller operating on 5bit integers will obviously be smaller and faster than its equivalent using 16bit encoding).

Since the dependence graph associated to the predicate evaluations is always acyclic (the predicates expressions only depends on the current loop indices values), all datapath operators bitwidths can be derived from the original loop indices bitwidths, which themselves depend on the loop bounds. Say $ub(i)$ is the upper bound for loop index i , its bitwidth is then given by $w_i = \lceil \log_2(ub(i)) \rceil$.

We can use parametric integer linear programming techniques to determine the upper bound for all the loop indices, for a given set of loop parameters values. For example, from the loop described in figure 4, the lower and upper bounds for indices k and j are $0, T$ and $0, N$ respectively. Their bitwidth is then given by $w_k = \lceil \log_2(N) \rceil$ and $w_j = \lceil \log_2(T) \rceil$ respectively.

Using this information, we then propagate the bitwidth constraint along the node of the DDG, using equation 1, in which in_1 and in_2 represent the graph node input port bitwidths, and f is the operation performed in the node at hand.

$$w(f, in_1, in_2) = \begin{cases} \max(w(in_1), w(in_2)) + 1 & \text{if } f = add \\ w(in_1) + w(in_2) & \text{if } f = mul \\ w(in_1) - w(in_2) + 1 & \text{if } f = div \\ w(in_2) & \text{if } f = mod \end{cases} \quad (1)$$

4.2 Example

We will now take a particular node of the the QR algorithm as an example and derive a parameterized predicated controller. Figure 4 shows the loop-iterators k and j and a number of predicates to activate the proper IPD operation, i.e., read from

a FIFO, and OPD operation, i.e., write to a FIFO. At each iteration of the loop-iterators, some predicates are evaluated to read the correct data from the correct FIFO. This is given by the READ part. Next the function `Func` is executed in the EXECUTE part. Finally data is written to the correct FIFO in the WRITE part.

```

0 void P2 ::main() {
1   for (int k = 1 ; j <= N ; j += 1 ) {
2     for (int j = 1 ; k <= T ; k += 1 ) {

3       if (k-2 >= 0)                                READ
4         in_0 = read(FIFO1);                          IPD_1
6       if (k-1 == 0)
7         in_0 = read(FIFO2);                          IPD_2
9       if (j-2 >= 0)
10        in_1 = read(FIFO3);                          IPD_3
12      if (j-1 == 0)
13        in_1 = read(FIFO4);                          IPD_4

13      (out_0, out_1, out_1) =                          EXECUTE
14        Func(in_0, in_1) ;

15      if (-k+T-1 >= 0)                                WRITE
16        write(out_0, FIFO1);                          OPD_1
17      if (-j+N-1 >= 0)
18        write(out_1, FIFO5);                          OPD_2
19      if (k-T == 0)
20        write(out_2, FIFO6);                          OPD_3

21    } // for k
22  } // for j

```

Figure 4: Process taken from the QR algorithm.

In the description given in Figure 4, each **read** (resp. **write**) operation is guarded by one or more linear expression predicates that are depending on the loop parameters and indices.

These linear expressions are first extracted from the loop-iterators along with the predicates, then, after applying common expression elimination, a data dependence graph (DDG) is constructed as given in Figure 4.2.

At the top of the DDG, 4 ports are shown. Two ports for the parameters N and K and two ports for the current loop-iterators k_t and j_t . The lower two port correspond to the values of k_{t+1} and j_{t+1} which will be fed back as inputs to the DDG after each iteration execution. Each time the DDG is evaluated, a particular switch pattern appears at the IPD and OPD outputs driving the Read and Write units in the virtual processor as shown in Figure 1.

Light gray nodes in the DDG correspond to operations that depends on the most inner loop index, and therefore need to be evaluated at each new iteration, dark gray nodes correspond to operations that depends on the most outer loop index, and therefore only need to be re-evaluated when the inner loop upper bound is reached.

To the difference of the ROM approach, the parameterized predicate controller implementation allows us to handle large parameterized iteration domains. We however observe two issues.

The first issue relates to the complexity of the domain (size,

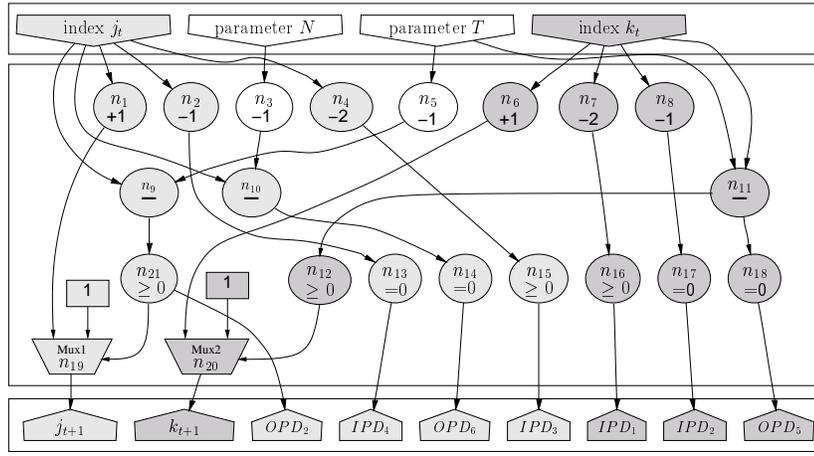


Figure 5: Data dependence graph associated to the loop described in figure 1.

shape) spanned by the loop-iterators and on the number of *IPDs/OPDs* involved in a node. Since all predicates and expressions have to be evaluated at every cycle, their evaluation requires more hardware resources as the domain gets more complex and the number of linear expression grows. As a consequence, for irregularly shaped domain, the controller implementation might use a lot of area.

A second issue relates to the fact that we map all predicates evaluation as pure combinational functions. As a consequence, the controller speed (i.e. the frequency at which the controller can run) might not be scalable: computational complexity (i.e its number of *terms* in the predicate expressions) usually increases with the number of dimension of the iteration domain. The resulting controller critical path will therefore be very dependent on the number of dimension of the domain.

5 Partitioned Parameterized Predicate Controller

The parameterized predicate controller is sensitive to the amount of linear expressions that need to be evaluated in each iteration. It is however possible to drastically reduce the amount of computations. To do so, we take advantage of the fact that only expression predicates that depend on the most inner loop index have to be evaluated at every cycle. Other linear expressions only need to be re-evaluated when one of their associated loop index changes value. This happens when one of the outer-loop iterators has reached its upper-bound.

5.1 Hardware structure of the controller

The partitioned parameterized predicate controller tries to take advantage of the dependence on loop-iterators to get a simplified controller that uses less hardware resources : As less expressions have to be evaluated, fewer hardware can be used to realize the controller. In the partitioned parameterized predicate controller, the controller is split into two parts : a parallel datapath and a sequential controller. The *parallel datapath* is similar to the one presented in Section 1.1. Its purpose is to

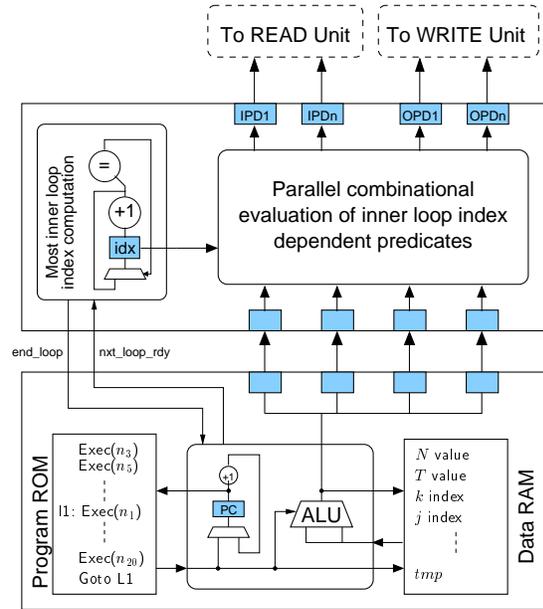


Figure 6: Partitioned hardware loop controller architecture

evaluate all expressions depending on the value of the *most inner loop-iterator*. The *sequential controller*, on the other hand, evaluates all the expressions that depend on the outer loop iterators and parameters (including the most inner loop upper and lower bounds) and forward its results to the parallel datapath.

In Figure 6, the parallel datapath is given in the top part and the sequential controller is given in the lower part. In the sequential controller, a small sequential program is stored in a Program ROM. When running this program on a simple micro-engine, some data values are computed and stored in the Data RAM. These values needed by the parallel datapath are forwarded to it. The parallel datapath evaluates at each cycle the parallel combinational logic, using the forwarded values. The only value that is re-computed, is the inner loop-iterator iteration. The partitioned parameterized predicate controller is gen-

erated in two steps.

5.1.1 DDG Partitioning

In the first step, we apply again common expression elimination and construct the data dependence graph as explained previously in Section 4.1.2. The main difference is that in this case we will partition this graph into a set of sub-graphs (one for each loop index). Each sub-graph contains all the nodes which have as input argument the sub-graph corresponding loop index. In case a node has two distinct loop-iterators as argument, we map the node to the sub-graph that is associated with the most inner loop-iterator.

5.1.2 Hardware mapping

In the next step, we partition the computation between the parallel and sequential controller. This partitioning uses the sub-graphs obtained in the DDG partitioning. All the nodes of the sub-graph associated to the most inner loop index are mapped on the parallel controller. All arguments that depend on values calculated by the sequential controller are mapped on communication ports with the sequential controller. The remaining sub-graphs are mapped on the sequential controller. We sequentially schedule the operations in the remaining sub-graphs, by performing a topological sort to ensure that data dependences constraints are satisfied. Then we generate the global sequential schedule by concatenating all these local schedules in a decreasing depth order. Values that need to be communicated to the parallel controller are mapped on communication ports with the parallel controller.

5.2 Example

As an example of how we derive a partitioned parameterized predicated controller, we look again at a node of the QR algorithm as given in Figure 4. The first step is to obtain the partitioned DDG. The DDG shown in figure 5 is therefore decomposed in three sub-graphs: G_{param} , G_j and, G_k , which are respectively associated to the parameters N and T and to the two loop indices j and k . In Figure 5, the color of a node indicates to which sub-graph it belongs to.

Next we need to map a sub-graph onto either the parallel controller or the sequential controller. Since j is the inner loop iterators, we map G_j onto the parallel controller. The two remaining sub-graphs are mapped onto the sequential controller. This means that the sequences of operations $V(G_p) = \{n_3, n_5\}$, and $V(G_k) = \{n_7, n_{16}, n_8, n_{17}, n_{11}, n_{18}, n_6, n_{12}, n_{20}\}$ are sequentialized to a program that runs on the sequential controller and which is shown in Figure 7.

5.3 Discussion

Some problems might appear when the most inner loop domain becomes very small (only a few iterations). In such a case, the sequential machine might not be able to compute the partial results needed by the inner-loop parallel datapath fast enough, therefore slowing-down the controller.

However, in most applications, processes very seldom need to fire at every cycle (either because they are in a blocking-read blocking write state), or because the IP core cannot start the execution of a new operation every cycle. The impact of such *hazards* is then very unlikely to impact the overall network performance.

```
// parameters
1 n3 := N-1;
2 write(n3,port1);
3 n5 := T-1;
4 write(n5,port2);
// k outer loop
5 n7 := (k-2);
6 n16 := n7>=0;
7 write(n16,port3);
8 n8 := (k-1);
9 n17 := n8=0
10 write(n17,port4);
11 n11 := (k-T);
12 n18 := n11=0;
13 write(n18,port5);
14 n6:= k+1;
15 n12 := n11>=0;
16 if n12
    n20:=n6;
    else
    n20:=1;
    end if;
17 k_next : =n20;
// sync with datapath
18 synchronize()
19 jump 3
```

Figure 7: A possible schedule for the sequential controller

5.4 Experimental results

To observe the benefits of our approach with respect to the ROM Table implementation, we used the same applications as in Table 1. For each of them we derived both a partitioned and non-partitioned parameterized predicate controller, which was then mapped on a Xilinx Virtex-2 FPGA. The results are given both in terms of frequency and resource usage (in FPGA *slices*) in Table 2. From these results, we can make the following remarks :

- The controller resource usage can vary a lot, depending on both the application and on the domain size (which influences the bitwidth of the operators in the datapath).
- It appears that the fixed area cost overhead caused by the sequential controller used in the partitioned approach is quite important, and makes this implementation strategy only viable for very large and complex domains.
- In all cases, the partitioned controller is slower than its counterpart. This slow-down is mostly due to the sequential controller, which suggest that its realization deserves more attention.

More generally, these two strategies provide very interesting results, even for very simple and small domains, and therefore suggest that the ROM table approach is not really appropriate

	Size		Non partitioned		Partitioned	
	N	T	MHz	Area	MHz	Area
QR factorization	8	16	140	29	100	112
	16	64	133	68	85	133
	64	256	121	89	74	163
	W	H	MHz	Area	MHz	Area
Stereo-vision	320	200	97	133	65	120
	640	400	100	148	74	123
	1024	640	100	153	71	126
	W	H	MHz	Area	MHz	Area
Optical-flow	320	200	129	97	76	98
	640	400	118	110	72	103
	1024	640	126	113	75	106

Table 2: Experimental Results for three representative examples.

whenever large iteration domains are involved (as it is the case for most image algorithms).

6 Run-Time Distance Evaluation Approach

6.1 Motivation

If you look to the code in Figure 4, you can observe that in most of the cases an IPD/OPD selection will remain active for a certain number of consecutive cycles. By taking advantage of this *regularity* it is possible to further reduce the control computational cost of a process. Instead of evaluating a set of predicates associated to an IPD/OPD at each cycle, we can compute for each IPD/OPD the number of cycles during which they will remain in the active state and load counters with these values (these values will later on be referred to as *pattern distances*). Then, at each firing of the process, these counters decrement, until they reach 0. They will then have to be reloaded with a new value corresponding to the next control computations.

The main problem with this approach is that we need one counter for each function argument. This is likely to be impractical in many cases since it is not rare in image processing algorithms to have functions with more than fifty arguments. Besides, scheduling the counter loading operations is also likely to be difficult. Therefore, we chose to merge the information of all the IPD/OPD active states into what we call *variant domains*, and to compute the *distances* corresponding to the number of cycles/iterations during which the process remains into the same variant domain.

6.2 Variant domains

For each process firing, the data is read/written from/to a configuration of active FIFO channels. Such a configuration together with the process function form a *process variant*. The notion of variant was introduced in [7]. With each such configuration corresponds a configuration of IPDs/OPDs. The polytope domain onto which this configuration remains unchanged is called *variant domain*. Having the polytope description of the IPDs and OPDs, the variant domains of a process are derived using the *DomainIntersection* function provided by PolyLib [20] and the Omega Test provided by the Omega library [12].

Consider a process function with n input/output arguments. For each function argument i there is a union L_i of disjoint polytope domains representing the active IPDs/OPDs that (depending on the iteration) deliver the data. We say L_i is the *argument list* corresponding to the function argument i . A variant domain L_i is therefore given by the intersection of components of an arbitrary element taken from the Cartesian product of the function argument lists:

$$(D_{i,1}, \dots, D_{i,n}) \in L_1 \times \dots \times L_n,$$

$$V_i = \bigcap_{t=1}^n D_{i,t} \leftarrow \text{DomainIntersection} \quad (2)$$

Please note that this resulting domain may be empty (such a domain corresponds to an IPD/OPD configuration that will never appear during the process execution). The emptiness of a variant domain is checked using the Omega Test:

$$V_i \cap Z \neq \emptyset \leftarrow \text{OmegaTest} \quad (3)$$

Following the procedure explained before, the construction of the variant domains for the Vectorize process of the QR algorithm is graphically depicted in Figure 8. Hence, the process from Figure 4 contains 9 different variants:

```
V1=(IPD2,IPD4,Vec,OPD1,OPD2,*) V6=(IPD1,IPD3,Vec,*,OPD2,OPD3)
V2=(IPD1,IPD4,Vec,OPD1,OPD2,*) V7=(IPD2,IPD3,Vec,OPD1,*,*)
V3=(IPD1,IPD4,Vec,*,OPD2,OPD3) V8=(IPD1,IPD3,Vec,OPD1,*,*)
V4=(IPD2,IPD3,Vec,OPD1,OPD2,*) V9=(IPD1,IPD3,Vec,*,*,OPD3)
V5=(IPD1,IPD3,Vec,OPD1,OPD2,*)
```

The “*” symbol present in some of the variant formats, means that the corresponding function argument, although produced, will not be distributed to any other processes in the network. Each variant is active inside the corresponding variant domain represented by a set of nested if-statements. Figure 9 shows the representation of the Vectorize process house-keeping code based on variant domains:

```
0 void P2 ::main() {
1   for (int j = 1 ; j <= N ; j += 1 ) {
2     for (int k = 1 ; k <= T ; k += 1 ) {
3       if ((k-1 == 0) && (j-1 == 0))
4         Execute (V1);
5       if ((k-2 >= 0) && (j-1 == 0) && (-k+T-1 >= 0))
6         Execute (V2);
7       if ((j-1 == 0) && (k-T == 0))
8         Execute (V3);
9       if ((k-1 == 0) && (j-2 >= 0) && (-j+N-1 >= 0))
10        Execute (V4);
11      if (( 2 <= k <= T-1) && (2<=j<=N-1))
12        Execute (V5);
13      if ((k-T == 0) && (2<=j<=N-1))
14        Execute (V6);
15      if ((k-1 == 0) && (j-N == 0))
16        Execute (V7);
17      if ((k-2 >= 0) && (j-N >= 0))
18        Execute (V8);
19      if ((k-T == 0) && (j-N == 0) )
20        Execute (V9);
21    } // for k
22  } // for j
```

Figure 9: A process implementation based on variant domains

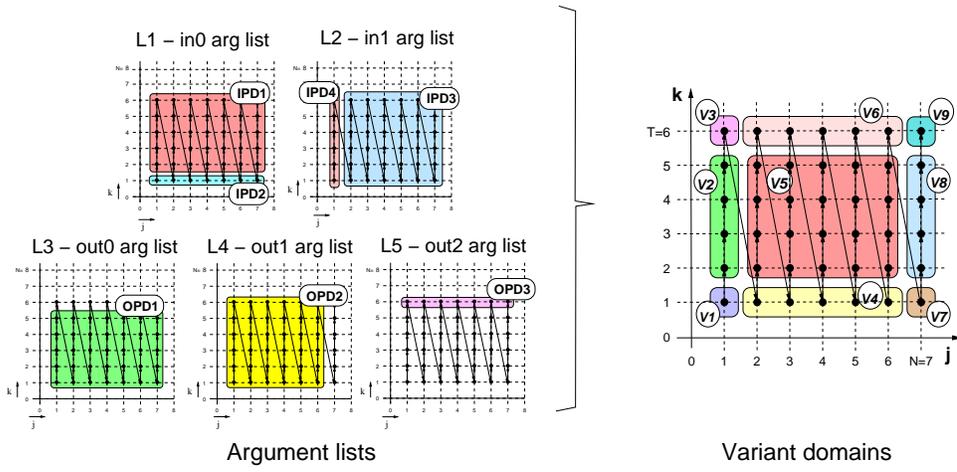


Figure 8: Deriving the variant domains corresponding to the Vectorize process

In general, for an arbitrary process the number of variants is greater or equal than the total number of input and output ports. On the other hand, by applying the technique presented in the previous section (common expression elimination) the complexity of the code stays the same i.e., the linear expressions predicates involved in the variant domain representation of a process are the same than those present in the description of the IPDs or OPDs. Hence, we can ensure the usage of the pattern distance approach without introducing extra control overhead.

6.3 Deriving the pattern distances

As explained in Section 6.1, we are trying to make use of the regularity existing in the IPD/OPD active configuration. The use of variant domains alone does not bring any improvement in terms of speed or control efficiency. However, a control realization based on variants is well suited to the *pattern distance approach* that was intuitively presented in Section 6.1.

In this subsection, we will present a formal way of expressing the regularity existent in the process variant description by expressing the loop control as a repetitive pattern of variant configurations. In other words, we seek to find the *pattern distance* (in terms of number of iteration) that separates distinct variant domains during the most inner loop execution. For this purpose we make use of the Ehrhart theory for computing the number of points inside parameterized polytope.

Definition 1 pattern distance: Let V be a n -dimensional variant domain and $V'(j) = \{x \in V \cap \mathbb{Q}^n \mid j = Prj(x) \wedge j \in Prj(V)\}$ a parameterized polytope, where Prj is a projection given by the matrix $[I_{n-1}, 0]$. The pattern distance $PD_V(j)$ is represented by the number of integral points inside the polytope $V'(j)$: $PD_V(j) = Card\{V'(j) \cap \mathbb{Z}^n\}$.

Because V is a polytope, $Prj(V)$ is also a polytope. Therefore, we can apply the parameterized version of the Ehrhart theorem to compute the number of integer points belonging to $V'(j)$. Because the projection matrix is $[I_{n-1}, 0]$, $V'(j)$ is a 1-dimensional polytope. As a consequence, PD_V is expressed by

a set of pseudo-polynomials $d_S(j)$ of degree one, each one of them valid inside a $(n - 1)$ -dimensional polytope Vd_S , called validity domain:

$$PD_V(j) = \begin{cases} d_1(j) & \text{if } j \in Vd_1, \\ \vdots & \\ d_S(j) & \text{if } j \in Vd_S. \end{cases} \quad (4)$$

For more information about the the Ehrhart theory we refer to [16, 1].

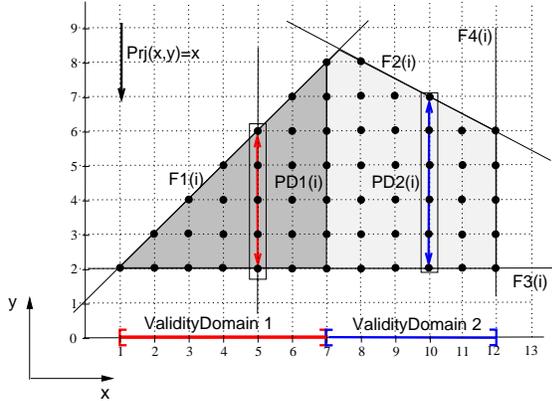


Figure 10: Deriving pattern distances

Example: Suppose a variant domain as given in Fig 10 represented by the polytope: $VD = \{(x, y) \in \mathbb{Q}^2 \mid 2 \leq y, y \leq x + 1, 2y \leq -x + 24\}$. Since VD is a 2-dimensional polytope, the projection function Prj is given by the mapping $M = [1, 0]$. According to this projection function, the parameterized polytope onto which we have to count the pattern distance is: $VD'(j) = \{(x, y) \in \mathbb{Q}^2 \mid 2 \leq y \leq x + 1, 2y \leq -x + 24, j = x, 1 \leq j \leq 12\}$. According to the Ehrhart theory, because $VD'(j)$ is a parameterized polytope, the number of integer points inside $VD'(j)$ is given by 2 pseudo-polynomial

expressions:

$$PD_{VD}(j) = \begin{cases} j & \text{if } j \in [1, 7], \\ -\frac{1}{2}j + [11, \frac{21}{2}]_j & \text{if } j \in [7, 12]. \end{cases} \quad (5)$$

6.4 Sequencing the pattern distances

Once the variant domains have been derived, and knowing that all of them lie in the same space, we can apply a similar matching technique as the one used for deriving the variant domains in Section 6.2. In this way, we obtain a set of polytope domains representing the *variant validity domains* of a process. To each variant validity domain VV corresponds a set of pattern distances PD_k , each one being computed inside of a specific variant domain VD_k . *In this way a variant validity domain (vvd) represents the process iteration space during which the pattern distances (corresponding to the active variants) stay the same.*

Since there is a one-to-one relation between the pattern distances and the variants active inside a vvd, we can uniquely define a pair $[V_n, PD_n]$. For a point $j \in VV$, each pattern distance $PD_k(j)$ represents the number of iterations during which the process stays into the same IPD/OPD configuration. Based on the lexicographic order we can find the order in which for a variant validity domain a process steps through the corresponding pattern distances. For this purpose we make use of parametric integer linear programming [3], namely to compute the lexicographical minimum point from an active variant domain:

Let $m(VD_i)$ be the lexicographical minimum element from each variant domain VD_i . Using the lexicographic order we induce a total order " δ " among the pairs $[V_n, PD_n]$ from a given vvd VD_i :

$$[V_m, PD_m(j)] \delta [V_n, PD_n(j)] \Leftrightarrow m(VD_m) < m(VD_n) \quad (6)$$

Thus, for each $j \in VV$ we can derive a sequence $\langle \dots, [V_1, PD_1(j)], \dots, [V_k, PD_k(j)], \dots \rangle$ made of pairs $[VariantDomain, PatternDistance]$, where $[V_1, PD_1(j)] \delta [V_k, PD_k(j)]$. Having the sequence of the pattern distances for all the variant validity domains we can restructure the process house keeping code for achieving fast execution time. Following the presented approach the process from Figure 4 becomes as follows:

```

0 void P2 ::main() {
1   for (int j = 1 ; j <= N ; j += 1 ) {
2     if ( j == 1 )
3       Execute <[V1,1], [V2, T-2], [V3,1]>;
4     if ( 2<= j <=N-1 )
5       Execute <[V4,1], [V5, T-2], [V6,1]>;
6     if ( j == N )
7       Execute <[V7,1], [V8, T-2], [V9,1]>;
8   } // for j

```

Figure 11: A process implementation based on pattern distances

As it can be seen, the Vectorize process contains 3 variant validity domains: $VV_1 = \{j \mid j = 1\}$, $VV_2 = \{j \mid 2 \leq j \leq N - 1\}$ and $VV_3 = \{j \mid j = N\}$. To each such

vvd corresponds a sequence of pairs $[VariantDomains, PatternDistances]$. For example for VV_2 there is the sequence $\langle [V_4, 1], [V_5, T - 2], [V_6, 1] \rangle$, which means that for a iteration points $j \in VV_2$ the process should fire one time the variant V_4 , then $(T - 2)$ times the variant V_5 , and finally once V_6 .

6.5 Hardware mapping

The hardware realization of the *distance-variant* based controller is similar to the partitioned approach described in Section 4.2. The controller is split in two sub-components (a pattern distance unit and a sequential controller unit) working in a producer-consumer manner as shown in Figure 12.

- The pattern distance unit consists of a loadable counter and of a variant decoder. This unit works as follows : When a new pattern distance is fetched from the FIFO, the distance value is loaded in the counter, and the variant is decoded into its associated IPD/OPD activity pattern. Then, at each firing, the counter decrements by one until it reaches 0 and fetches a new pattern distance from the FIFO.
- The sequential controller computes the sequence of pattern distances associated with the process, and forwards them to the FIFOs.

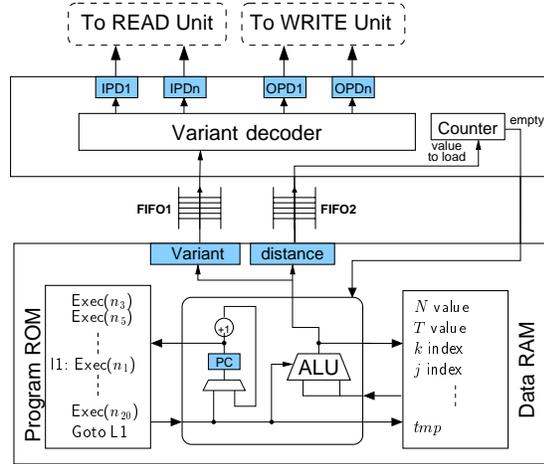


Figure 12: Distance based controller architecture

7 Discussion and Future Work

In this paper we have investigated three different approaches for efficiently deriving the control part involved in routing data through each KPN process generated by Compaan/Laura. The abstract representation of a Compaan process is given in terms of polyhedra, such that the work presented here is related to the work done in the context of scanning union of polyhedra using for-loops [14, 2].

From the experimental results, it turns out that in order to derive the optimal controller implementation, the characteristics

of the application (like domain size and shape, loop nest dimension, number of function arguments or number of IPDs/OPDs) have to be taken into consideration. From the different approaches proposed in this paper, two of them (ROM and parametric predicate) are currently automated by Laura. Future research considering the other two approaches should hence be considered. Since the range of applications accepted by Compaan is wide we will also investigate the opportunity of having a strategy for choosing the appropriate approach depending the application characteristics.

REFERENCES

- 1 Ph. Clauss and V. Loechner. Parametric analysis of polyhedral iteration spaces. *Journal of VLSI Signal Processing*, 19:179–194, July 1998.
- 2 J.F. Collard, P. Feautrier, and T. Risset. Construction of DO loops from systems of affine constraints. *Parallel Processing Letters*, 5:421–436, 1995.
- 3 Paul Feautrier. Parametric integer programming. *Operations Research*, 22(3):243–268, 1988.
- 4 Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.
- 5 Peter Held. *Functional Design of Data-Flow Networks*, 1996. PhD thesis, Delft University of Technology, The Netherlands.
- 6 Gilles Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*. North-Holland Publishing Co., 1974.
- 7 A.C.J. Kienhuis. *Design Space Exploration of Stream-based Dataflow Architectures: Method and Tools*. PhD thesis, Delft University of Technology, January 1999.
- 8 Bart Kienhuis. MatParser: An array dataflow analysis compiler. Technical report, University of California at Berkeley, 2000. UCB/ERL M00/9.
- 9 Bart Kienhuis, Edwin Rijpkema, and Ed F. Deprettere. Compaan: Deriving process networks from matlab for embedded signal processing architectures. In *8th International Workshop on Hardware/Software Codesign (CODES'2000)*, San Diego, USA, May 2000.
- 10 Kurt Konolige. Small vision systems: Hardware and implementation. In *Eighth International Symposium on Robotics Research*, Hayama, Japan, October 1997.
- 11 E. Mémin and T. Risset. On the study of VLSI derivation for optical flow estimation. *International Journal of pattern recognition and Artificial Intelligence (IJPRAI)*, 2000.
- 12 W. Pugh. The omega test: A fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 35(8):102–114, 1992.
- 13 William Pugh and David Wonnacott. An Exact Method for Analysis of Value-based Array Data Dependences. In *Proceedings of the Sixth Workshop on Programming Languages and Compilers for Parallel Computing*, Dec 93.
- 14 F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5):469,498, 2000.
- 15 Edwin Rijpkema. *From Piecewise Regular Algorithms to Dataflow Architectures*. PhD thesis, Delft University of Technology, 2001.
- 16 Richard P. Stanley. *Enumerative Combinatorics: Volume 1*. Wadsworth Inc., Belmont, California 94002, 1986.
- 17 Todor Stefanov, Claudiu Zissulescu, Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. System design using kahn process networks: The compaan/laura approach. In *Proceedings of DATE2004*, Paris, France, Feb 16 – 20 2004.
- 18 Alexandru Turjan, Todor Stefanov, Bart Kienhuis, and Ed Deprettere. The compaan tool chain: Converting matlab into process networks. In *In Designers' Forum "Design, Automation and Test in Europe (DATE 2002)"*, Paris, France, 2002.
- 19 R. L. Walke, R. W. M. Smith, and G. Lightbody. 20GFLOPS QR processor on a Xilinx Virtex-e FPGA. In *proceedings of SPIE advanced signal*, 1999.
- 20 Doran K. Wilde. A library for doing polyhedral operations. Master's thesis, Oregon State University, Corvallis, Oregon, Dec 1993. Also published in IRISA technical report PI 785, Rennes, France; Dec, 1993.
- 21 Claudiu Zissulescu, Todor Stefanov, Bart Kienhuis, and Ed Deprettere. Laura: Leiden architecture research and exploration tool. In *Proceedings of the 13th International Conference on Field Programmable Logic and Applications (to appear)*, Lisbon, Portugal, September 2003.