Adaptive streaming applications : analysis and implementation models

Zhai, J.T.

Chapter 6

A New MoC for Modeling Adaptive Streaming Applications

Jiali Teddy Zhai, Hristo Nikolov, Todor Stefanov, "Modeling Adaptive Streaming Applications with Parameterized Polyhedral Process Networks", *In the Proceedings of the 48th IEEE/ACM Design Automation Conference (DAC'11)*, pp. 116–121, San Diego, CA, USA, June 5-9, 2011.

THE popular parallel MoCs for streaming applications are compared in Figure 1.6 on page 10 in terms of expressiveness against compile-time analyzability. For example, models such as SDF, CSDF, and PPN are fairly popular due to their design-time analyzability. However, they have the limitation of allowing only static parameters, whose values are fixed at design-time and they can not be changed at run-time. As a consequence, adaptive streaming applications cannot be expressed using these MoCs.

In contrast, the general MoCs shown in Figure 1.6 include BDF, SADF, KPN, and RPN. They provide capability of modeling adaptive application behavior. However, these general models are not analyzable at design-time. Therefore, we are interested in a model which is able to capture adaptive/dynamic behavior in applications while allowing design-time analyzability to some extent. In this context, Parameterized SDF/CSDF (PSDF/PCSDF) and FSM-SADF models have been proposed as extensions of the SDF/CSDF models. However, scenario reconfiguration in FSM-SADF is limited to a set of pre-defined scenarios. For PSDF/PCSDF, a complex consistency check and computing schedules have to be performed at run-time.

To overcome these issues, in this chapter we introduce a parameterized extension of the PPN model, called Parameterized Polyhedral Process Networks (P^3N). P^3N

improves the expressiveness of PPN, allowing to model adaptive streaming applications. Compared to the aforementioned PSDF/PCSDF and FSM-SADF models, P^3N allows more flexible parameter reconfiguration and enables efficient techniques (less complex) for run-time consistency check by performing part of the consistency check at design-time. In the Daedalus^{RT} design flow, the P^3N MoC is used as the implementation model similar to the PPN MoC for static streaming applications.

6.1 Related Work

In [94], a mathematical model and semantics for reconfiguration of dataflow models are proposed. This approach analyzes where and how parameter values can be changed dynamically and consistently according to dependence relations between parameters. Our P^3N model provides similar semantics for reconfiguration. In particular, for P^3Ns, it is possible to extract dependence relations between dependent parameters at design-time, which is not discussed in [94].

In PSDF/PCSDF [29], separate *init* and *sub-init* graphs are proposed to reconfigure *body* graphs in a hierarchical manner. In the PSDF/PCSDF models, for every combination of parameter values, both computing a schedule and verifying consistency need to be resolved at run-time. In contrast, our P^3N model does not require computing schedules at run-time because all processes are self-scheduled based on the KPN semantics. Therefore, at run-time, only the consistency check has to be performed. The consistency check is furthermore facilitated by the efficient approach we have devised (and present further in this chapter) to extract relations between dependent parameters at design-time.

In SADF [114] and FSM-SADF [47], *detector* actors are introduced to parameterize the SDF model. All valid scenarios must be pre-defined at design-time. Each scenario consists of a set of valid parameter combination that determines a *scenario* of SADF. This guarantees the consistency of SADF in individual scenarios, therefore, no run-time consistency check is required. In a scenario, the SADF model behaves the same ways as the SDF model. Therefore, an SADF graph can be seen as a set of SDF graphs. In the initial FSM-SADF definition, all the production and consumption rates of the dataflow edges are constant within a graph iteration of a scenario. Recently in [49], an extension, called weak consistency, has been made to FSM-SADF. A weakly-consistent FSM-SADF graph allows scenario changes within a graph iteration of a scenario. For P^3N, no prior knowledge of valid parameter combinations is assumed, as the run-time consistency check (see Section 6.3) will guarantee consistency of the P^3N model. We consider that this flexibility is desired compared to FSM-SADF. Once a P^3N model is reconfigured, it behaves as a PPN model. Therefore, a P^3N can be seen as a set of PPNs. Production and consumption

patterns in the P^{3}N model thus may still vary during the execution of a particular parameter configuration[1].

For all parameterized models discussed above, the performance penalty due to reconfiguration of parameters at run-time has never been evaluated when these models are executed on MPSoC platforms. This can be an important factor in determining the metric of implementation efficiency (see Figure 1.6(b)) while comparing two adaptive models. In contrast, in this chapter we study the performance penalty introduced by the run-time consistency check and the reconfiguration of our P^{3}Ns on real MPSoC implementations.

6.2 Model Definition

Consider the example of a P^{3}N given in Figure 6.1(c) and a non-parameterized PPN in Figure 6.1(a). Although the dataflow topology of the P^{3}N is the same as the PPN, processes P_2 and P_3 are parameterized by two parameters M and N which values are updated by the environment at run-time using process Ctrl and edges E_7, E_8, E_9. PPN process P_3 is shown in Figure 6.1(b) and P^{3}N Process P_3 is shown in Figure 6.1(d). We use this example throughout the chapter. Below, we formally define the P^{3}N model.

6.2.1 Parameterized Polyhedral Process Networks

Definition 6.2.1 (Parameterized Polyhedral Process Network)**.** A Parameterized Polyhedral Process Network (P^{3}N) is defined by a graph $G = (\mathcal{P}, P_{ctrl}, \mathcal{E})$, where

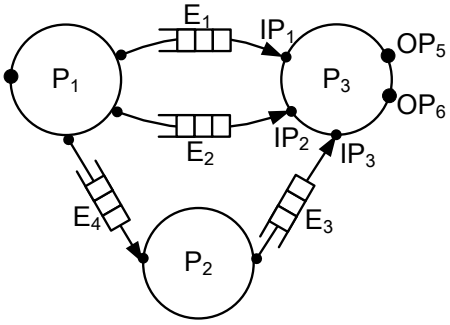
- $\mathcal{P} = \{P_1, ..., P_{|\mathcal{P}|}\}$ is a set of dataflow processes,
- P_{ctrl} is the control process,
- $\mathcal{E} = \{E_1, ..., E_{|\mathcal{E}|}\}$ is a set of edges, which are FIFOs.

For the P^{3}N shown in Figure 6.1(c), $\mathcal{P} = \{P_1, P_2, P_3\}$ is the set of dataflow processes. Process Ctrl is the control process P_{ctrl}. $\mathcal{E} = \{E_1, E_2, E_3, E_4, E_7, E_8, E_9\}$ is the set of edges, which are FIFOs.

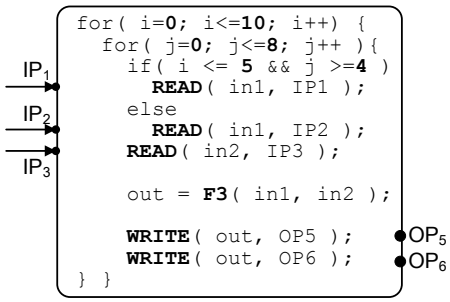
Definition 6.2.2 (Dataflow Process)**.** A dataflow process P is described by a tuple $(\mathcal{I}_P, \mathcal{O}_P, F_P, D_P)$, where

- $\mathcal{I}_P = \{IP_1, ..., IP_{|\mathcal{I}_P|}\}$ is a set of input ports,
- $\mathcal{O}_P = \{OP_1, ..., OP_{|\mathcal{O}_P|}\}$ is a set of output ports,

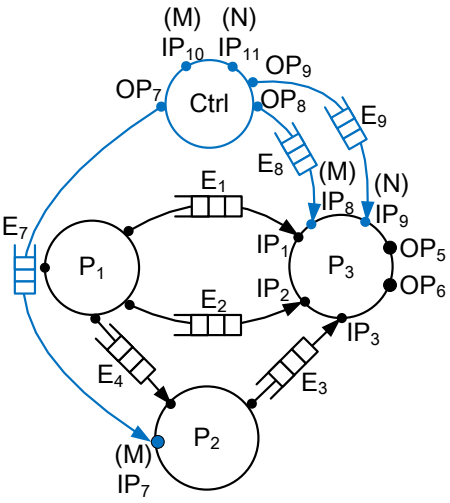
[1]This is called a *process cycle* in Definition 6.2.8 and it is equivalent to a scenario in the SADF model.



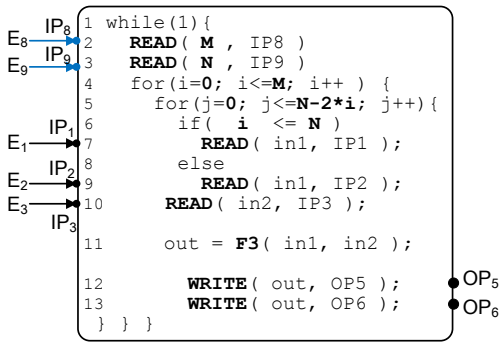
(a) A PPN.



(b) PPN process P_3.



(c) A P^{3}N.



(d) P^{3}N process P_3.

Figure 6.1: Comparsion between a PPN and a P^{3}N.

- F_P is the process function defined by a tuple $(M_P, ARG_{in}, ARG_{out})$, where ARG_{in} and ARG_{out} are sets of variables and $M_P : ARG_{in} \rightarrow ARG_{out}$ is a mapping relation,
- D_P is the process domain defined by a parametric polyhedron (see Definition 2.1.3 on page 24).

In Figure 6.1(d), dataflow process P_3 has input ports $I_{P_3} = \{IP_1, IP_2, IP_3, IP_8, IP_9\}$ and output ports $O_{P_3} = \{OP_5, OP_6\}$. Process function $F_3 = (\texttt{F3}, \{\texttt{in1}, \texttt{in2}\}, \texttt{out})$ maps variables `in1` and `in2` to variable `out` with process function `F3`. Assume that the range of parameters M and N is bounded by the polytope (see Definition 2.1.2

on page 23) $\bar{\mathcal{D}}_{(M,N)}$ as

$$\bar{\mathcal{D}}_{(M,N)} = \{(M,N) \in \mathbb{Z}^2 \mid 0 \leq M \leq 100 \wedge 0 \leq N \leq 100\},$$

then the process domain of P_3 is represented as a parametric polyhedron

$$D_{P_3}(M,N) = \{(w,i,j) \in \mathbb{Z}^3 \mid w > 0 \wedge 0 \leq i \leq M \wedge 0 \leq j \leq N - 2i\}.$$

Definition 6.2.3 (Input Port)**.** An input port IP of process P is described by a tuple (V, D_{IP}), where

- V is a variable which:
  - binds the port to process function F_P if $V \in ARG_{in}$;
  - binds the port to process domain D_P or other port domains D_{IP}, D_{OP} if $V \in \vec{p}$, where $\vec{p}$ is the parameter vector defined in Definition 2.1.3 on page 24,
- D_{IP} is the input port domain defined by a parametric polyhedron, where $D_{IP} \subseteq D_P$.

Definition 6.2.4 (Output Port)**.** An output port OP of process P is described by a tuple (V, D_{OP})

- V is a variable which binds the port to process function F_P if $V \in ARG_{out}$,
- D_{OP} is the output port domain defined by a parametric polyhedron, where $D_{OP} \subseteq D_P$.

In Figure 6.1(d), input port IP_1 of process P_3 is defined as $IP_1 = (\texttt{in1}, D_{IP_1})$, where

$$D_{IP_1}(M,N) = \{(w,i,j) \in \mathbb{Z}^3 \mid w > 0 \wedge 0 \leq i \leq M \wedge i \leq N \wedge 0 \leq j \leq N - 2i\}.$$

Similarly, output port OP_5 is defined as $OP_5 = (\texttt{out}, D_{OP_5})$, where OP_5 is bound to variable out and $D_{OP_5}(M,N) = D_3(M,N)$.

Definition 6.2.5 (Control Process)**.** A control process P_{ctrl} is described by a tuple $(I_{ctrl}, F_{ctrl}, O_{ctrl}, D_{ctrl})$, where

- $I_{ctrl} = \{(\perp, p_1, D_{IP}), ..., (\perp, p_m, D_{IP})\}$ is a set of input ports.
- F_{ctrl} is the process function defined by a tuple $(Eval, \{\vec{p}, \vec{p}_{old}\}, \vec{p}_{new})$, where $\vec{p}$, $\vec{p}_{old}$ and $\vec{p}_{new}$ are parameter vectors. $Eval : (\vec{p}, \vec{p}_{old}) \rightarrow \vec{p}_{new}$ is the specific mapping relation discussed in Sections 6.2.2 and 6.3.

- $O_{ctrl} = (V, D_{OP}), ..., (V, D_{OP})$ is a set of output ports, where $V \in \vec{p}_{new}$.
- D_{ctrl} is the process domain, where $D_{ctrl} = D_{IP} = D_{OP} = \{w \in \mathbb{Z} \mid w > 0\}$.

Control process Ctrl of the P^{3}N shown in Figure 6.1(c), is given in Figure 6.2(a). Its structure and behavior are discussed in Section 6.2.2 in detail.

Definition 6.2.6 (Edge)**.** An edge $E \in \mathcal{E}$ is defined by a tuple

$$\Big((P_i, OP_k), (P_j, IP_l)\Big)$$

where

- P_i is the process that writes data to edge E through output port OP_k.
- P_j is the process that reads data from edge E through input port IP_l.

In P^{3}Ns, the process domain and port domains are formally defined as parametric polyhedrons (see Definition 2.1.3 on page 24), which allows for mathmatical analysis and manipulation. The polyhedral representation of P^{3}N can be easily converted to sequential nested-loop programs [25] and vice versa [126]. Thus, for the sake of clarity, we present processes in the form of sequential programs in the examples of this chapter.

6.2.2 Operational Semantics

The processes in our P^{3}N MoC execute autonomously and communicate via FIFOs obeying the KPN semantics, which is similar to the PPN MoC. In this section, we formally define our additional, specific operational semantics of the P^{3}N MoC that makes it different from the PPN MoC.

Definition 6.2.7 (Process Iteration)**.** A process iteration of process P is a point $(w, x_1, ..., x_d) \in D_P$, where the following operations are performed sequentially:

1. reading one token from each IP if $(w, x_1, ..., x_d) \in D_{IP}$.
2. executing process function F_P.
3. writing one token to each OP if $(w, x_1, ..., x_d) \in D_{OP}$.

In process P_3 shown in Figure 6.1(d), a process iteration (lines 6-13) consists of reading one token for variable `in1` from either input port IP_1 or IP_2, one token for variable `in2` from input port IP_3, executing process function `F3`, and writing one token for variable `out` to output ports OP_5 and OP_6.

Definition 6.2.8 (Process Cycle)**.** The ith process cycle $CYC_P(i, \vec{p_i}) \in D_P$ of a process P is a set of lexicographically ordered process iterations. It is expressed as a polytope

$$CYC_P(i, \vec{p_i}) = \{(w, x_1, ..., x_d) \in \mathbb{Z}^{d+1} \mid A \cdot (w, x_1, ..., x_d)^T \geq B \cdot \vec{p_i} + b \wedge w = i\},$$

where $i \in \mathbb{Z}^+$ and $\vec{p_i} \in \bar{\mathcal{D}}^P_{\vec{p}} \subseteq \bar{\mathcal{D}}_{\vec{p}}$.

Definition 6.2.9 (Process Execution)**.** Process execution EX_P is a sequence of process cycles denoted by

$$CYC_P(1, \vec{p}_1) \leftarrow CYC_P(2, \vec{p}_2) \leftarrow ... \leftarrow CYC_P(i, \vec{p}_i),$$

where $i \rightarrow \infty$ and $\vec{p}_i \in \bar{\mathcal{D}}^P_{\vec{p}}$.

Overall, every process in a P^{3}N executes on indefinite number of process cycles in accordance with Definition 6.2.9. For instance, $CYC_{P_3}(2,(7,8))$ denotes the second process cycle that corresponds to the execution of the nested *for*-loops (lines 4-13) when $(M,N) = (7,8)$ during the execution of process P_3 given in Figure 6.1(d).

In the P^{3}N model, parameters in dataflow processes can change values during the execution, i.e., $\vec{p}_i \neq \vec{p}_{i+1}$. Thus, it is necessary to define the operational semantics related to changing of parameter values. Similar to quiescent points in [94], we also define the points at which changing the value of $\vec{p}$ is permitted.

Definition 6.2.10 (Quiescent Point of a Dataflow Process)**.** A point

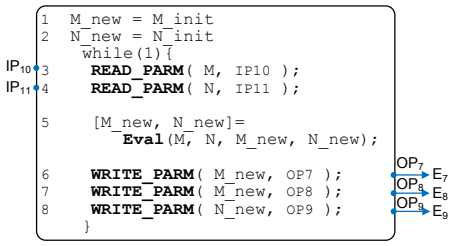
$$Q_P(i, \vec{p_i}) \in CYC_P(i, \vec{p_i})$$

of dataflow process P is a quiescent point if $CYC_P(i, \vec{p_i}) \in EX_P$ and it satisfies

$$\neg(\exists(w, x_1, ..., x_d) \in CYC_P(i, \vec{p_i}) \ : \ (w, x_1, ..., x_d) \prec Q_P(i, \vec{p_i}))$$

According to Definition 6.2.10, dataflow processes can change parameter values at the first process iteration of any process cycle during the execution. For instance, process P_3 given in Figure 6.1(d) updates parameters (lines 2-3) before executing the nested *for*-loops in every process cycle. Generally, updating parameters at each quiescent point is initiated by reading from edges which are connected to the control process.

The control process plays an important role in the P^{3}N's operational semantics. It reads parameter values from the environment and propagates only valid parameter values to the dataflow processes. Valid parameter values lead to consistent execution of P^{3}Ns (see Section 6.3). The validity of the parameter values is evaluated by



(a) Control process Ctrl in Figure 6.1(c).

```
[ M_new, N_new ]
Eval(M, N, M_old, N_old){

  // checking parameters
  par_ok = Check(M, N);

  if( par_ok ){
    return (M, N);
  else {
    return(M_old, N_old);
} }
```

(b) Function `Eval`.

Figure 6.2: Control process and evaluation function.

process function *Eval* defined in Definition 6.2.5. The control process sends the latest parameter combination that has been evaluated as valid, which means that P^{3}Ns always respond to changes of the environment as fast as possible. Also, the dataflow processes need to read the parameter values in the correct order. Therefore, to keep the same order of parameter values for all dataflow processes, the control process writes to the control edges, e.g., edges E_7, E_8 and E_9 in Figure 6.1(c), only when all control edges have at least one buffer space available. Here the control edges are implemented as non-bloking-write FIFOs. In case that any of these FIFOs is full, the incoming parameter combination is discarded and the control process continue to read the next parameter combination from the environment. Furthermore, the depth of the FIFOs of the control edges determines how many process cycles of the dataflow processes are allowed to overlap.

Let us consider the P^{3}N shown in Figure 6.1(c). The behavior of the control process is given in Figure 6.2(a). Process Ctrl starts with at least one valid parameter combination (lines 1-2) and then reads parameters from the environment (lines 3-4) repetitively. For every incoming parameter combination, the process function `Eval` (line 5) checks whether the combination of parameter values is valid. The implementation of function `Eval` is given in Figure 6.2(b). In Section 6.3, we present details about the implementation of function `Check`. If the combination is valid, then function `Eval` returns the current parameter values (`M, N`). Otherwise, the last valid parameters combination (propagated through `M_new, N_new` in this example) is returned. After the evaluation of the parameter combination, process Ctrl writes the parameter values to output ports (lines 6-8) when all edges E_7, E_8, and E_9 have at least one buffer space available.

6.3 Consistency

As defined in Section 6.2, P^{3}Ns operate on input streams with infinite length. Thus, the P^{3}Ns, we are interested in, must be able to execute without deadlocks and only using FIFOs with finite capacity. This kind of P^{3}Ns is considered to be *consistent*. In this section, we first define the consistency condition of the P^{3}N model and then present an approach to preserve the consistent execution of P^{3}Ns at run-time.

Definition 6.3.1 (Consistency of a P^{3}N)**.** A P^{3}N is consistent if

$$\forall E = ((P_i, OP_k), (P_j, IP_l))$$

and $k \to \infty$, it satisfies

$$|D^{CYC}_{OP_k}| = |D^{CYC}_{IP_l}|,$$

where

$$D^{CYC}_{OP} = CYC_{P_i}(c, \vec{p}_s) \cap D_{OP_k},$$
$$D^{CYC}_{IP} = CYC_{P_j}(c, \vec{p}_t) \cap D_{IP_l},$$

$CYC_{P_i}(c, \vec{p}_s) \in EX_{P_i}$, and $CYC_{P_j}(c, \vec{p}_t) \in EX_{P_j}$.

Consider edge E_3 connecting processes P_2 and P_3 of the P^{3}N given in Figure 6.1(c). The execution of processes P_2 and P_3 is illustrated in Fig. 6.3. The access of both processes to edge E_3 is depicted in Figure 6.4. Definition 6.3.1 requires that, for each corresponding process cycle of both processes $CYC_{P_2}(i, M_i)$ and $CYC_{P_3}(i, M_i, N_i)$, the number of tokens $|D^{CYC}_{OP_3}(M)|$ produced by process P_2 to edge E_3 must be equal to the number of tokens $|D^{CYC}_{IP_3}(M, N)|$ consumed by process P_3 from edge E_3.

It is not trivial to preserve the consistent execution of a P^{3}N as defined in Definition 6.3.1. First of all, at each quiescent point Q_P during the execution of a process, the incoming parameter values $\vec{p}_s$ and $\vec{p}_t$ are unknown at design-time, which may result in different $|D^{CYC}_{OP_k}|$ and $|D^{CYC}_{IP_l}|$ at run-time for any edge E connecting dataflow processes. Therefore, whether a P^{3}N can be executed consistently with a given parameter combination, has to be checked at run-time. Secondly, computing $|D^{CYC}_{OP_k}|$ and $|D^{CYC}_{IP_l}|$ is challenging as well. Below, we demonstrate the difficulties associated with checking the consistency using edge E_3 given in Figure 6.4 as an example. One question that naturally arises is which combinations of (M, N) ensure the consistency condition as defined by Definition 6.3.1. For instance, if $(M, N) = (7, 8)$, P_2 produces 25 tokens to E_3 and P_3 consumes 25 tokens from the same edge after one corresponding process cycle of both processes. It can be verified that P_2

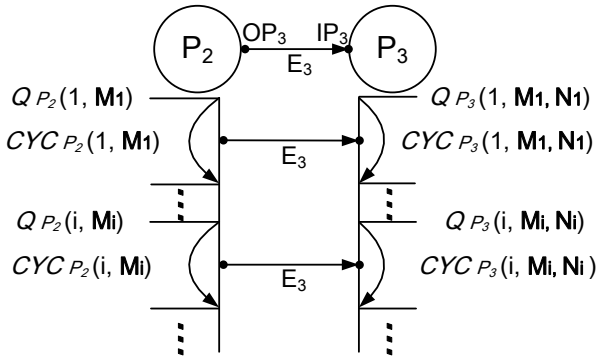


Figure 6.3: Consistent execution of process P_2 and P_3 w.r.t. edge E_3.

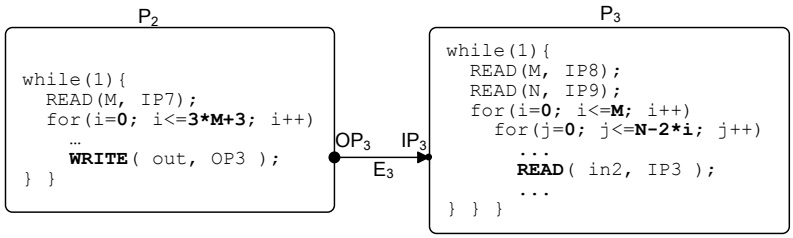


Figure 6.4: Which combinations (M,N) do ensure consistency of P^{3}N?

produces 13 tokens to E_3 while P_3 requires 20 tokens from it if $(M,N) = (3,7)$ in a corresponding process cycle. Thereby, in order to complete one execution cycle of P_3 in this case, it will read data from E_3 which will be produced during the next execution cycle of P_2. Evidently this leads to an incorrect execution of the P^{3}N. From this example, we can clearly see that the incoming values of (M,N) must satisfy certain relation to ensure the consistent execution of the P^{3}N.

Although the consistency of a P^{3}N has to be checked at run-time, still some analysis can be done at design-time. First, from Definition 6.3.1, we can see that both D_{OP}^{CYC} and D_{IP}^{CYC} are parametric polytopes. We can check the condition $|D_{OP}^{CYC}| = |D_{IP}^{CYC}|$ by comparing the number of integer points in both parametric polytopes D_{OP}^{CYC} and D_{IP}^{CYC}. This is thus equivalent to computing cardinality of both D_{OP}^{CYC} and D_{IP}^{CYC}. In this work, we use the *Barvinok* library [127] to compute cardinality of a parametric polytope. The *Barvinok* library can solve the problem in polynomial time. In general, the number of integer points inside a parametric polytope is defined as a list of (quasi-)polynomials (see Definition 2.1.4 on page 25). A quasi-polynomial is a polynomial with periodic numbers as coefficients. For instance, considering

input port IP_3 shown in Figure 6.4, $D^{CYC}_{IP_3}$ is given as

$$D^{CYC}_{IP_3}(M,N) = \{(i,j) \in \mathbb{Z}^2 \mid 0 \leq i \leq M \wedge 0 \leq j \leq N - 2i\}.$$

The number of tokens $|D^{CYC}_{IP_3}(M,N)|$ read by function `READ(in2, IP3)` in one process cycle is represented as the list of polynomials found by the *Barvinok* library:

$$\begin{cases} 1+N+N\cdot M-M^2 & \text{if } (M,N)\in C1 \\ 1+\frac{3}{4}N+\frac{1}{4}N^2+\frac{1}{4}N-\frac{1}{4}\cdot\{0,1\}_n & \text{if } (M,N)\in C2 \end{cases} \tag{6.1}$$

where $C1$ and $C2$ are called *chambers* (see Definition 2.1.4 on page 25) given as

$$C1 = \{(M,N) \in \mathbb{Z}^2 \mid M \leq N \wedge 2M \geq 1+N\},$$
$$C2 = \{(M,N) \in \mathbb{Z}^2 \mid 2M \leq N\}.$$

In addition, the second polynomial is a quasi-polynomial, in which $\{0,1\}_n$ is a periodic coefficient with period 2. For instance, function `READ(in2, IP3)` reads $1+\frac{3}{4}\times 7+\frac{1}{4}\times 7^2+\frac{1}{4}\times 7-\frac{1}{4}\times 1 = 20$ tokens in one process cycle if $(M,N) = (3,7) \in C2$. Below, we present the approach we have devised to extract all parameter combinations that satisfy the consistency condition defined in Definition 6.3.1. Algorithm 8 summarizes the analysis we performed at design-time. Recall that the condition $|D^{CYC}_{OP}| = |D^{CYC}_{IP}|$ must be satisfied for a consistent execution of a P^{3}N. Thus, for each edge connecting dataflow processes, we first compute $|D^{CYC}_{OP}|$ and $|D^{CYC}_{IP}|$. Two lists of (quasi-)polynomials are obtained. If a P^{3}N can execute consistently with a certain parameter combination, individual (quasi-)polynomials in both lists must be equal. We check the equivalence by subtracting the (quasi-)polynomials from both lists symbolically. The symbolic subtraction can result in zero, a non-zero constant, or (quasi-)polynomial. If the result is zero, the consistency is always preserved for all parameters within the range of chamber C_{res}. At run-time, these parameters are propagated immediately to destination dataflow processes. If a non-zero constant is obtained, all parameters within the range of chamber C_{res} are discarded at run-time, because these parameter values would break the consistency condition of the resulting P^{3}N. In the third case, the result is a (quasi-)polynomial in which only some parameter combinations within the range of chamber C_{res} are valid for the consistency condition. We provide two alternatives to extract all valid parameter combinations within this range by solving the resulting equation $q_{res}(\vec{p_{jt}}) = 0$. In the first alternative, the equation can be solved at design-time against all possible parameter combinations. A table, which contains all solutions, i.e., all valid parameter combinations, is generated and stored in function *Check*. At run-time, the control process only propagates those incoming parameter combinations

Algorithm 8: Generation of polynomials for function *Check*

Input: A P^{3}N
Result: A list of (quasi-)polynomials
1 **foreach** *edge E corresponding* (OP_{P_O}, IP_{P_I}) **do**
2 Compute $|D_{OP}^{CYC}|$ and $|D_{IP}^{CYC}|$ using the *Barvinok* library;
3 **foreach** *(quasi-)polynomial* $q_{OP}(\vec{p_j})$ *in* $|D_{OP}^{CYC}|$ **do**
4 Get chamber C ;
5 **foreach** *(quasi-)polynomial* $q_{IP}(\vec{p_t})$ *in* $|D_{IP}^{CYC}|$ **do**
6 Get chamber C' ;
7 Compute $q_{res}(\vec{p_{jt}}) = q_{OP}(\vec{p_j}) - q_{IP}(\vec{p_t})$;
8 Compute chamber $C_{res} = C \cup C'$;
9 **if** $q_{res}(\vec{p_{jt}}) = 0$ **then**
10 Consistency is preserved for chamber C_{res};
11 **else if** $q_{res}(\vec{p_{jt}})$ *is a non-zero constant* **then**
12 Eliminate chamber C_{res};
13 **else**
14 Store (quasi-)polynomial $q_{res}(\vec{p_{jt}})$ with C_{res} ;

that match an entry in the table. In the second alternative, function *Check* evaluates $q_{res}(\vec{p_{jt}})$ against zero with incoming parameter values at run-time.

Let us consider the example shown in Figure 6.4 again. We apply Algorithm 8 to extract the valid parameter combinations. Besides $|D_{IP_3}^{CYC}(M,N)|$ as given in Equation (6.1), $|D_{OP_3}^{CYC}(M)| = 3M+4$ is obtained. Subtraction of the (quasi-)polynomials in $|D_{OP_2}^{CYC}(M)|$ and $|D_{IP_2}^{CYC}(M,N)|$ yields two $q_{res}(M,N)$:

$$\begin{cases} (1+N+N\cdot M-M^2)-(3M+4)=0 & \text{if } (M,N)\in C1 \\ (1+\frac{3}{4}N+\frac{1}{4}N^2+\frac{1}{4}N-\frac{1}{4}\cdot\{0,1\}_n)-(3M+4)=0 & \text{if } (M,N)\in C2 \end{cases} \quad (6.2)$$

where chambers $C1$ and $C2$ are equal to the chambers in Equation (6.1). Clearly this corresponds to the third case in Algorithm 8 (see line 14). The structure of the two alternatives of function `Check` is given in Figures 6.5(a) and 6.5(b). The solutions to Equation (6.2) stored in table `tab` is shown in Figure 6.5(a), whereas evaluating Equation (6.2) directly against zero at run-time is depicted in Figure 6.5(b). In this example, if the range of the parameters is $0 \leq M,N \leq 100$, then there are only 10 valid parameter combinations. In addition, if $0 \leq M,N \leq 1000$, the valid

```
[ par_ok ]
Check ( M , N ) {
  tab =  [ (4, 6),
           (7, 8),
           (15, 12),
            …  ];

  // found M , N in tab
  if ( found )
    return par_ok = true;
  else
    return par_ok = false;
}
```

(a) First alternative.

```
[ par_ok ]
Check ( M , N ) {
  // chamber C1
  if ( M <= N && 2M >= N+1 )
    res = -M^2 + N *M - 3M + N -3;

  // chamber C2
  if ( 2M <= N  )
    res = N^2/4 + 3N/4 + (N%2)/2 - 3M -3;

  if ( res == 0)
    return(true);
  else
    return(false);
}
```

(b) Second alternative.

Figure 6.5: Two alternatives of Function `Check` in Figure 6.2(b).

number of parameter combinations are 34, and if $0 \leq M,N \leq 10000$, the number of combinations increases to 114.

6.4 Experimental Results

In order to evaluate the run-time overhead introduced by the reconfiguration of our P^{3}N model, in this section, we present the results we have obtained by mapping a P^{3}N onto a Xilinx Virtex 6 FPGA platform. We have selected a synthetic P^{3}N with complex quasi-polynomials in order to quantify the performance penalty caused by evaluating complex quasi-polynomials at run-time. In order to measure the run-time reconfiguration overhead, we have also implemented the reference PPNs. These PPNs contain only the dataflow processing of the corresponding P^{3}N. The experiments have been conducted using the ESPAM tool and the Xilinx Platform Studio (XPS) 13.2 tool. The generated MPSoCs consist of several MicroBlaze (MB) soft-core processors connected using Xilinx' Fast Simplex Link (FSL) FIFOs. To avoid additional execution overhead, in these experiments, every process has been mapped onto a separate MB processor.

The P^{3}N we consider is depicted in Fig. 6.6. It is formed by the processes in Figure 6.4 and one additional process P_4. Figure 6.6 also shows the representation of processes P_3 and P_4 in order to show the domains $D^{CYC}_{OP_5}(M,N)$ and $D^{CYC}_{IP_5}(N)$ of ports OP_5 and IP_5, connected to edge E_5. Consequently, applying Algorithm 8 yields the following two polynomials for edge E_5:

$$\begin{cases} (1+N+N\cdot M-M^2)-(3N+1)=0 & \text{if } (M,N)\in C1 \\ (1+\frac{3}{4}N+\frac{1}{4}N^2+\frac{1}{4}N-\frac{1}{4}\cdot\{0,1\}_n)-(3N+1)=0 & \text{if } (M,N)\in C2 \end{cases} \tag{6.3}$$

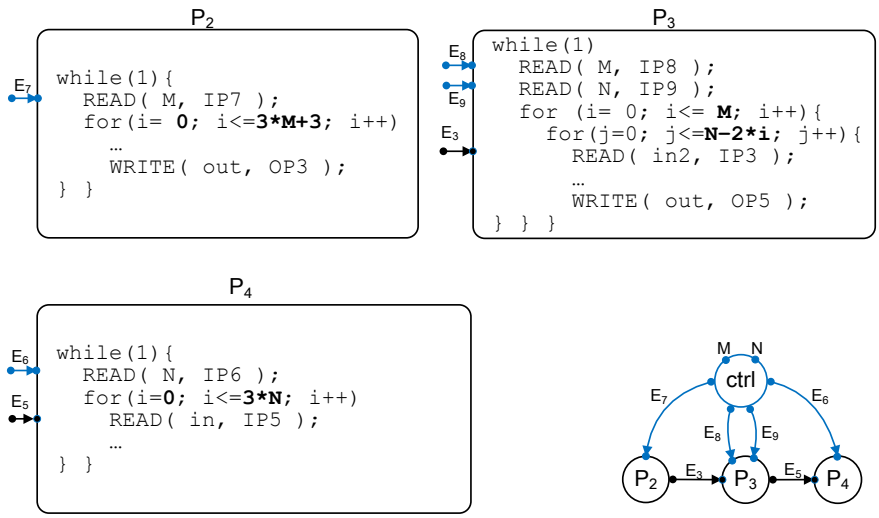


Figure 6.6: P^{3}N of our experiment

where

$$C1 = \{(M,N) \in \mathbb{Z}^2 \mid M \leq N \wedge 2M \geq 1+N\},$$
$$C2 = \{(M,N) \in \mathbb{Z}^2 \mid 2M \leq N\}.$$

For edge E_3, the dependence relation of parameters M and N is already given in Equation (6.2). In a first implementation alternative, we solved Equations 6.2 and 6.3 at design-time and stored all possible parameter values that have been found in a table into function *Check* of control process ctrl. In a second implementation alternative, the polynomials in Equations 6.2 and 6.3 have been evaluated directly in function *Check* at run-time. Furthermore, we have configured five different workloads of the dataflow processes by gradually increasing the execution latency of processes P_2, P_3, and P_4. We have run the MPSoC implementations on an FPGA board for 10 different valid parameter combinations, i.e., process ctrl reconfigures the dataflow processes 10 times within parameter range $0 \leq M,N \leq 100$.

For the P^{3}Ns which evaluate the polynomials at run-time (the third bar of each configuration), we have made the following observations. First, configurations 1 and 2 show a relatively large overhead. This is because these configurations correspond to the situation where the execution latency of processes P_2, P_3, and P_4 is very small. That is, the dataflow processes are very light-weight, therefore, they are mostly blocked on reading from the control edges in order to update values of parameters M and N. In this way, configurations 1 and 2 give a good indication about the time needed to evaluate the polynomials. Second, if we increase the execution latency

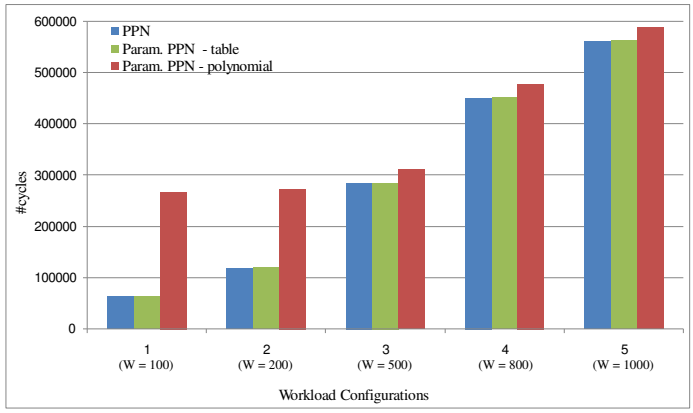


Figure 6.7: Performance results of PPN and P^{3}N implementations

of the dataflow processes, then the introduced overhead is significantly reduced, see configurations 3, 4, and 5 in Figure 6.7. In these three configurations, the overhead is only 9%, 5%, and 4%, respectively. In addition, we have observed that the absolute values of the overhead (in clk cycles) stay constant. This is because in these three configurations, the dataflow completely overlaps with the evaluation of the polynomials. We have found that the difference with the reference PPN is caused by i) the time for the first evaluation of the polynomials at the beginning of the P^{3}N execution, i.e., in the beginning no overlap is possible, and ii) the time to read the parameter values from the control edges, i.e., such reading is not present in the reference PPNs. This is an important observation because it shows that the run-time reconfiguration of the P^{3}N model can be very efficient. Moreover, in most real-life streaming applications, a process execution latency is large enough to cancel out the overhead caused by the evaluation of the polynomials. For example, a discrete cosine transform (used in JPEG encoders) implemented on a MB processor requires a couple of thousand of clk cycles. Therefore, we conclude that the introduced run-time overhead is reasonable considering the more expressive power that the P^{3}N model provides than other models.