Modeling Adaptive Streaming Applications with Parameterized Polyhedral Process Networks

Jiali Teddy Zhai, Hristo Nikolov, Todor Stefanov
Leiden Institute of Advanced Computer Science,
Leiden University,
The Netherlands
{tzhai, nikolov, stefanov}@liacs.nl

ABSTRACT

The Kahn Process Network (KPN) model is a widely used model-of-computation to specify and map streaming applications onto multiprocessor systems-on-chips. In general, KPNs are difficult to analyze at design-time. Thus a special case of the KPN model, called Polyhedral Process Networks (PPN), has been proposed to address the analyzability issue. However, the PPN model is not able to capture adaptive/dynamic behavior. Such behavior is usually expressed by using parameters which values are reconfigured at run-time. To model the adaptive/dynamic applications, in this paper we introduce an extension of the PPN model, called *Parameterized Polyhedral Process Networks* (P^3N), which still provides design-time analyzability to some extent. We first formally define the P^3N model and its operational semantics. In addition, we devise a design-time analysis to extract relations between parameters. Based on the analysis, we propose an approach to ensure that consistent execution of the P^3N model is preserved at run-time. Using an FPGA-based MPSoC platform, we present a performance evaluation of the possible overhead caused by the run-time reconfiguration.

Categories and Subject Descriptors

F.1.1 [**Theory of Computation**]: Models of Computation; I.6.4 [**Simulation and Modeling**]: Model Validation and Analysis

General Terms

Design, Theory, Verification

Keywords

Model of computation, adaptive embedded systems, verification

1. INTRODUCTION

With the rapid increase of the complexity in multiprocessor system-on-chip (MPSoC) designs, as well as in the applications these systems execute, the traditional design process at a low-level of abstraction becomes very error-prone and time-consuming. Rising the abstraction level of the design process to electronic system level (ESL) [1] seems to be an inevitable way to increase the design productivity. In this respect, models of computation (MoC) play a crucial role for the success of the ESL concept. That is, almost all existing ESL approaches and tools rely on MoCs as a generalized way to describe the system behavior. As a result, analyzing and verifying desired properties of a given system specification can be performed in a systematic and automated way. In addition, specifying an application using a parallel MoC, usually as a set of concurrent application tasks with a well defined mechanism for inter-task communication and synchronization, facilitates the MPSoC programming. The parallel MoCs can be roughly categorized into process-based and dataflow models. For example, models such as Synchronous Dataflow (SDF) [13] and Cyclo-Static Dataflow (CSDF) [5] are fairly popular due to their design-time analyzability. Kahn Process Networks (KPN) [12] is another popular MoC, which in contrast to SDF/CSDF, is a process-based model. Although researches have indicated that the KPN model is suitable to specify and map streaming applications targeting MPSoC platforms [7, 18, 11, 21], in general, analyzing KPNs is not possible at design-time. Thus several special cases of the KPN model have been been proposed [19, 14, 9]. Among them, the Polyhedral Process Network (PPN) model [19] has the following advantages. That is, buffer sizes [19] and throughput [16] are decidable at design-time as well as automated HW/SW synthesis from PPNs is possible [11]. Moreover, PPNs can be automatically derived from sequential nested loop programs by the `pn` compiler [20].

Nowadays, many streaming applications in the domain of multimedia, image, and signal processing employ adaptive algorithms. For example, a computer vision system processes different parts of an image continuously to obtain information from several regions of interest depending on the actions taken by the external environment. It is often desirable that the system should still continue behaving correctly even though the input action is incorrect. Usually, the adaptive behavior is captured by using parameters, which values need to be updated at run-time. We call such parameters dynamic parameters and their values are not known at design-time. Models such as SDF/CSDF and PPN have the limitation of allowing only static parameters. The values of the static parameters are fixed at design-time and they can not be changed at run-time. As a consequence, the adaptive behavior is not amenable to the models such as SDF/CSDF and PPN.

To address the abovementioned problem, more expressive models are needed. For example, general models such as Boolean-controlled Dataflow [6], Dynamic Data Flow [6], KPN [12], and Reactive KPN [10] 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) [4] and Scenario-aware Dataflow (SADF) [3] models have been proposed as extensions of the SDF/CSDF models. However, the expressiveness of adaptivity in SADF is limited. For PSDF/PCSDF, a complex consistency check and computing schedules have to be performed at run-time.

To overcome these issues, in this paper 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. Com-

DAC'11, June 5-10, 2011, San Diego, California, USA

pared to the aforementioned PSDF/PCSDF and SADF models, P^{3}N has higher expressive power and enables efficient techniques (less complex) for run-time consistency check by performing part of the consistency check at design-time.

1.1 Paper Contributions

In this paper, we introduce the Parameterized Polyhedral Process Networks (P^{3}N) model and define its operational semantics which allows for flexible update of parameter values at run-time. In addition, we propose a consistency check approach which is applied at both, design-time and run-time. Based on the P^{3}N semantics, we have devised a design-time approach to extract relations between parameters if they are dependent. This leads to a consistent parameterization of the model and moreover, it simplifies the run-time consistency check. Finally, we show that the P^{3}N model is suitable for targeting MPSoC implementations. A quantitative evaluation of the overhead due to run-time change of parameter values is performed on an FPGA-based MPSoC platform.

1.2 Related Work

In [15], a general 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^{3}N model provides similar semantics for reconfiguration. In particular, for P^{3}Ns, it is possible to extract dependence relation between dependent parameters at design-time, which is not discussed in [15].

In PSDF/PCSDF [4], 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^{3}N 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 paper) to extract relations between dependent parameters at design-time.

In SADF [3], *detector* actors are introduced to parameterize the SDF model. In SADF, all scenarios are explicitly specified by valid parameter combinations. This guarantees the consistency of the model, therefore, no run-time consistency check is required. In addition, all the production and consumption rates of the dataflow channels are constant within a scenario/configuration. In contrast, the P^{3}N does not have such a restriction. That is, production and consumption patterns in the P^{3}N model may still vary during the execution of a particular configuration. This makes the P^{3}N model more expressive than SADF.

For all parameterized models discussed above, the overhead due to reconfiguration of parameters at run-time has never been evaluated when these models are executed on MPSoC platforms. In contrast, in this paper we study the overhead introduced by the run-time consistency check and the reconfiguration of our P^{3}Ns on real MPSoC implementations.

The remaining part of the paper is organized as follows. Section 2 formally defines the P^{3}N model and its operational semantics. In Section 3, we present the approach we have devised to ensure consistent parameterization of P^{3}Ns. Section 4 quantitatively evaluates performance overhead caused by the reconfiguration and the consistency check performed at run-time. Section 5 concludes this paper.

2. FORMAL DEFINITION AND OPERATIONAL SEMANTICS

In this section, we formally define the P^{3}N model (Sec. 2.2) and its operational semantics (Sec. 2.3). Since the P^{3}N model is an

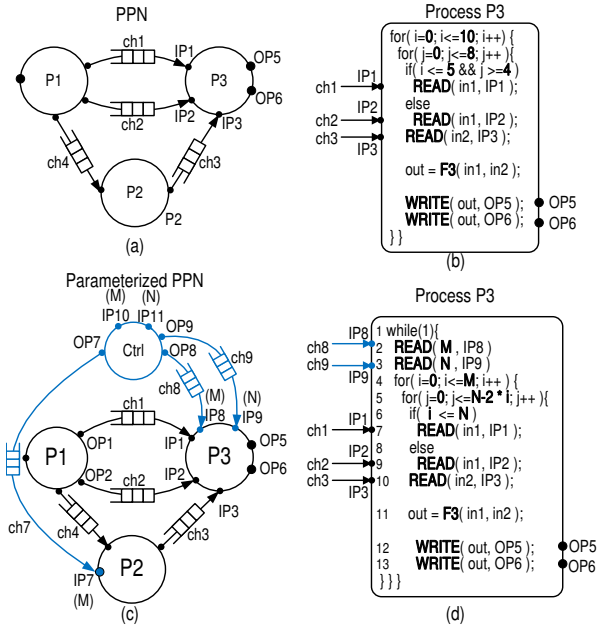

Figure 1: (a) An example of a PPN, (b) process *P3* in the PPN, (c) an example of a P^{3}N, and (d) process *P3* in the P^{3}N.

extension of the PPN model, to better understand the P^{3}N model, we first introduce the PPN model with an example.

2.1 Preliminaries – the PPN MoC [19]

A PPN consists of several autonomous processes communicating via FIFO channels. The PPN model is a special case of the KPN model because the processes in PPNs are structured and execute in a particular way. That is, a process first reads data from FIFO channels, then executes a function (computational behavior), and writes results to FIFO channels. Processes are synchronized based on the KPN semantics, i.e., any process is blocked when attempting to read an empty FIFO. Meanwhile the PPN model assumes finite FIFO buffers. Therefore, processes also block when attempting to write to a full FIFO. The `pn` compiler [20] computes a safe buffer size of each channel that guarantees the absence of deadlock in the network. The processes in the PPN model are represented in the polytope model [8]. Formally, a polytope $\mathcal{P}$ is defined as $\mathcal{P} = \{\vec{x} \in \mathbb{Z}^d \mid A \cdot \vec{x} \geq b\}$. An example of a PPN with processes *P1*, *P2*, and *P3* is depicted in Fig. 1(a). Process *P3*, shown in Fig. 1(b), first reads data from input ports to initialize variable *in1* and *in2*. After executing function *F3*, process *P3* writes the result *out* to output ports. The execution of process *P3* is specified by a two-dimensional polytope, $\{(i,j) \in \mathbb{Z}^2 \mid 0 \leq i \leq 10 \wedge 0 \leq j \leq 8\}$. The execution of read and write primitives is a subset of the polytope guarded by *if*-conditions. For instance, process *P3* reads input data from port *IP1* when $(i,j) \in \{(i,j) \in \mathbb{Z}^2 \mid 0 \leq i \leq 5 \wedge 4 \leq j \leq 8\}$.

2.2 Parameterized Polyhedral Process Networks

An example of a P^{3}N is given in Fig. 1(c). Although its dataflow topology is the same as the PPN in Fig. 1(a), process *P2* and *P3* are reconfigured by two parameters M and N which values are updated by the environment at run-time using process *Ctrl* and FIFO channels *ch7*, *ch8*, *ch9*. Process *P3* is shown in Fig. 1(d). We use this example throughout the paper. Below, we formally define the P^{3}N model.

Definition 1. A **Parameterized Polyhedral Process Network (P^{3}N)** is defined by a tuple $(\mathscr{P}, P_{ctrl}, \mathscr{E})$, where

- $\mathscr{P} = \{P_1, \ldots, P_J\}$ is a set of dataflow processes,
- P_{ctrl} is the control process,

- $\mathscr{E} = \{Ch_1, \ ..., \ Ch_H\}$ is a set of FIFO channels.

For the P^{3}N shown in Fig. 1(b), $\mathscr{P} = \{P1, P2, P3\}$ is the set of dataflow processes. Process *Ctrl* is the control process P_{ctrl}. $\mathscr{E} = \{ch1, ch2, ch3, ch4, ch7, ch8, ch9\}$ is the set of FIFO channels.

Definition 2. A **dataflow process** P is described by a tuple (I_P, O_P, F_P, D_P), where

- $I_P = \{IP_1, \ ..., \ IP_K\}$ is a set of input ports,
- $O_P = \{OP_1, \ ..., \ OP_L\}$ is a set of output ports,
- 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.

Definition 3. A **parametric polyhedron** $\mathcal{P}(\vec{\mathbf{p}})$ [19] is a polyhedron $\mathcal{P}$ affinely depending on a parameter vector $\vec{\mathbf{p}} = (\mathbf{p_1}, \ ..., \ \mathbf{p_m})^T$, i.e., $\mathcal{P}(\vec{\mathbf{p}}) = \{(w, x_1, ..., x_d) \in \mathbb{Z}^{d+1} \mid A \cdot (w, x_1, ..., x_d)^T \geq B \cdot \vec{\mathbf{p}} + b\}$, where $\vec{\mathbf{p}}$ is bounded by a polytope $\mathcal{P}_{\vec{\mathbf{p}}} = \{\vec{\mathbf{p}} \in \mathbb{Z}^m \mid C \cdot \vec{\mathbf{p}} \geq d\}$. A bounded polyhedron is called a polytope.

In Fig. 1(d), dataflow process *P3* has input ports $I_{P3} = \{IP1, IP2, IP3, IP8, IP9\}$ and output ports $O_{P3} = \{OP5, OP6\}$. Process function $F_{P3} = (F3, \{in1, in2\}, out)$ maps variables *in1* and *in2* to variable *out* with mapping relation *F3*. Assume that the range of parameters M and N is bounded by the polytope $\mathcal{P}^{P3}_{(\mathbf{M,N})} = \{(M, N) \in \mathbb{Z}^2 \mid 0 \leq M \leq 100 \wedge 0 \leq N \leq 100\}$, then the process domain of *P3* is represented as a parametric polyhedron $D_{P3}(M, N) = \{(w, i, j) \in \mathbb{Z}^3 \mid w > 0 \wedge 0 \leq i \leq M \wedge 0 \leq j \leq N - 2i\}$.

Definition 4. An **input port** *IP* of process P is described by a tuple (*CH*, V, D_{IP}), where

- *CH* is the FIFO channel connected to the port if *CH* $\in \mathscr{E}$; *CH* is the environment to which the port is connected if *CH* $= \perp$,
- 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 $V \in \vec{\mathbf{p}}$;
- D_{IP} is the input port domain defined by a parametric polyhedron, where $D_{IP} \subseteq D_P$.

Definition 5. An **output port** *OP* of process P is described by a tuple (*CH*, V, D_{OP})

- *CH* is the FIFO channel connected to the port if *CH* $\in \mathscr{E}$; *CH* is the environment to which the port is connected if *CH* $= \perp$,
- 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 Fig. 1(d), input port *IP1* of process *P3* is defined as *IP1* $= (ch1, in1, D_{IP1})$, where $D_{IP1}(M, N) = \{(w, i, j) \in \mathbb{Z}^3 \mid w > 0 \wedge 0 \leq i \leq min(M, N) \wedge 0 \leq j \leq N - 2i\}$. Similarly, output port *OP5* is defined as *OP5* $= (\perp, out, D_{OP5})$, where *OP5* is connected to the environment through variable *out* and $D_{OP5}(M, N) = D_{P3}(M, N)$.

Definition 6. A **control process** P_{ctrl} is described by a tuple (I_{ctrl}, F_{ctrl}, O_{ctrl}, D_{ctrl}), where

- $I_{ctrl} = \{(\perp, \mathbf{p_1}, D_{IP}), ..., (\perp, \mathbf{p_m}, D_{IP})\}$ is a set of input ports.
- F_{ctrl} is the process function defined by a tuple (*Eval*, $\{\vec{\mathbf{p}}$, $\vec{\mathbf{p}}_{\mathbf{old}}\}$, $\vec{\mathbf{p}}_{\mathbf{new}}$), where $\vec{\mathbf{p}}$, $\vec{\mathbf{p}}_{\mathbf{old}}$ and $\vec{\mathbf{p}}_{\mathbf{new}}$ are parameter vectors. *Eval* : $(\vec{\mathbf{p}}, \vec{\mathbf{p}}_{\mathbf{old}}) \rightarrow \vec{\mathbf{p}}_{\mathbf{new}}$ is the specific mapping relation discussed in Sec. 2.3 and Sec. 3.
- $O_{ctrl} = (Ch_1, V, D_{OP}), ..., (Ch_i, V, D_{OP})$ is a set of output ports, where $V \in \vec{\mathbf{p}}_{\mathbf{new}}$.
- 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 Fig. 1(c), is given in Fig. 2(a). Its structure and behavior are discussed in Sec. 2.3 in detail.

Definition 7. A **channel** *Ch* $\in \mathscr{E}$ is defined by a tuple (O_{Ch}, I_{Ch}), where

- O_{Ch} is given by a tuple (P_O, OP_{Ch}), where P_O is the process that writes data to channel *Ch* through output port OP_{Ch},
- I_{Ch} is a given by a tuple (P_I, IP_{Ch}), where P_I is the process that reads data from channel *Ch* through input port IP_{Ch}.

In P^{3}Ns, the process domain and port domains are formally defined as parametric polyhedrons, which allows for formal, mathmatical analysis and manipulation. The polyhedral representation can be easily converted to sequential nested-loop programs and vice versa [2]. Thus, for the sake of clarity, we present processes in the form of sequential programs in the examples of this paper.

2.3 Operational Semantics

In general, the processes in the P^{3}N model execute autonomously and communicate via FIFO channels obeying the KPN semantics. In this section, we formally define the additional, specific operational semantics of the P^{3}N model that makes it different from the general KPN model.

Definition 8. A **process iteration** of process P is a point $(w, x_1, ..., x_d) \in D_P$, where the following operations are performed sequentially: reading one token from each IP if $(w, x_1, ..., x_d) \in D_{IP}$, executing process function F_P, and writing one token to each OP if $(w, x_1, ..., x_d) \in D_{OP}$.

In process *P3* shown in Fig. 1(d), a process iteration (lines 6-13) consists of reading one token for variable *in1* from either input port *IP1* or *IP2*, one token for variable *in2* from input port *IP3*, executing process function *F3*, and writing one token for variable *out* to output ports *OP5* and *OP6*.

Definition 9. Given two vectors $\vec{a}, \vec{b} \in \mathbb{Z}^n$, $\vec{a} \prec \vec{b}$ denotes that $\vec{a}$ is lexicographically smaller than $\vec{b}$, iff

$$\bigvee_{i=1}^{n} \left(a_i < b_i \wedge \bigwedge_{j=1}^{i-1} a_j = b_j\right)$$

Definition 10. A **process cycle** $CYC_P(\mathcal{S}, \vec{\mathbf{p}_i}) \in D_P$ is a set of lexicographically ordered process iterations. It is expressed as a polytope $CYC_P(\mathcal{S}, \vec{\mathbf{p}_i}) = \{(w, x_1, ..., x_d) \in \mathbb{Z}^{d+1} \mid A \cdot (w, x_1, ..., x_d)^T \geq B \cdot \vec{\mathbf{p}}_i + b \wedge w = \mathcal{S}\}$, where $\mathcal{S} \in \mathbb{Z}^+$ and $\vec{\mathbf{p}}_i \in \mathcal{P}^P_{\vec{\mathbf{p}}}$.

Definition 11. **Process execution** E_P is a sequence of process cycles denoted by $CYC_P(1, \vec{\mathbf{p}}_1) \leftarrow CYC_P(2, \vec{\mathbf{p}}_2) \leftarrow ... \leftarrow CYC_P(i, \vec{\mathbf{p}}_i)$, where $i \rightarrow \infty$ and $\vec{\mathbf{p}}_i \in \mathcal{P}^P_{\vec{\mathbf{p}}}$.

Overall, every process in a P^{3}N executes infinite number of process cycles in accordance with Def. 11. For instance, $CYC_{P3}(2, (7, 8))$ 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 *P3* given in Fig. 1(d).

In the P^{3}N model, parameters in dataflow processes can change values during the execution, i.e., $\vec{\mathbf{p}}_i \neq \vec{\mathbf{p}}_{i+1}$. Thus, it is necessary to define the operational semantics related to changing of parameter values. Similar to quiescent points in [15], we also define the points at which changing values of $\vec{\mathbf{p}}$ is permitted.

Definition 12. A point $Q_P(\mathcal{S}, \vec{\mathbf{p}_i}) \in CYC_P(\mathcal{S}, \vec{\mathbf{p}_i})$ of dataflow process P is a **quiescent point** if $CYC_P(\mathcal{S}, \vec{\mathbf{p}_i}) \in E_P$ and $\neg(\exists (w, x_1, ..., x_d) \in CYC_P(\mathcal{S}, \vec{\mathbf{p}_i}) : (w, x_1, ..., x_d) \prec Q_P(\mathcal{S}, \vec{\mathbf{p}_S}))$.

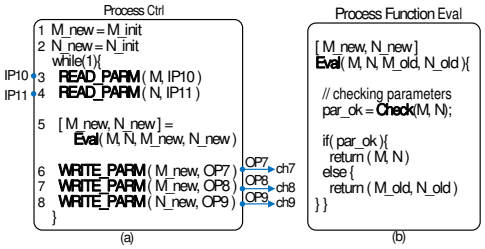

Figure 2: (a) Control process *Ctrl* and (b) process function *Eval*.

According to Def. 12, dataflow processes can change parameter values at the first process iteration of any process cycle during the execution. For instance, process *P3* given in Fig. 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 FIFO channels 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 Sec. 3). The validity of the parameter values is evaluated by process function *Eval* defined in Def. 6. 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 the control channels, e.g., channels *ch7*, *ch8*, *ch9* in Fig. 1(c), only when all control channels have at least one location available. In case that any of these FIFOs is full, the incoming parameter combination is discarded and the control process continues to read the next parameter combination from the environment. Furthermore, the depth of the FIFOs of the control channels determines how many process cycles of the dataflow processes are allowed to overlap.

Let us consider the P^{3}N shown in Fig. 1(c). The behavior of the control process *Ctrl* is given in Fig. 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) every a pre-specified time interval. 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 Fig. 2(b). In Sec. 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 channels *ch7, ch8*, and *ch9* have at least one location available.

3. CONSISTENCY

As defined in Sec. 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 13. A P^{3}N is **consistent** if $\forall Ch = ((P_O, OP), (P_I, IP)), i \rightarrow \infty, \Rightarrow |D_{OP}^{CYC}| = |D_{IP}^{CYC}|$, where $D_{OP}^{CYC} = CYC_{P_O}(i, \vec{\mathbf{p}}_j) \cap D_{OP}$, $D_{IP}^{CYC} = CYC_{P_I}(i, \vec{\mathbf{p}}_t) \cap D_{IP}$, P_O and P_I are dataflow processes, $CYC_{P_O}(i, \vec{\mathbf{p}}_j) \in E_{P_O}$, and $CYC_{P_I}(i, \vec{\mathbf{p}}_t) \in E_{P_I}$.

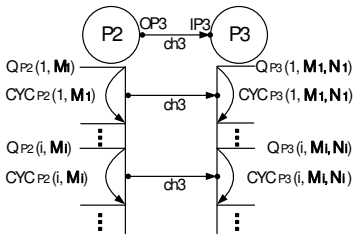

Figure 3: Consistent execution of process *P2* and *P3* w.r.t. channel *ch3*.

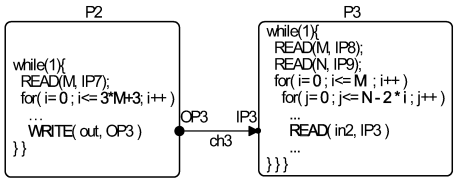

Figure 4: Which combinations $(M,\ N)$ do ensure consistency of P^{3}N?

Consider channel *ch3* connecting processes *P2* and *P3* of the P^{3}N given in Fig. 1(c). The execution of processes *P2* and *P3* is illustrated in Fig. 3. The access of both processes to channel *ch3* is depicted in Fig. 4. Def. 13 requires that, for each corresponding process cycle of both processes $CYC_{P2}(i, M_i)$ and $CYC_{P3}(i, M_i, N_i)$, the number of tokens $|D_{OP3}^{CYC}(M)|$ produced by process *P2* to channel *ch3* must be equal to the number of tokens $|D_{IP3}^{CYC}(M, N)|$ consumed by process *P3* from channel *ch3*.

It is not trivial to preserve the consistent execution of a P^{3}N as defined in Def. 13. First of all, at each quiescent point Q_P during the execution of a process, the incoming parameter values $\vec{\mathbf{p}}_j$ and $\vec{\mathbf{p}}_t$ are unknown at design-time, which may result in different $|D_{OP}^{CYC}|$ and $|D_{IP}^{CYC}|$ at run-time for any channel *Ch* 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 $|D_{OP}^{CYC}|$ and $|D_{IP}^{CYC}|$ is challenging as well. Below, we demonstrate the difficulties associated with checking the consistency using channel *ch3* given in Fig. 4 as an example. One question that naturally arises is which combinations of (M, N) ensure the consistency condition as defined by Def. 13. For instance, if $(M, N) = (7, 8)$, *P2* produces 25 tokens to *ch3* and *P3* consumes 25 tokens from the same channel after one corresponding process cycle of both processes. It can be verified that *P2* produces 13 tokens to *ch3* while *P3* requires 20 tokens from it if $(M, N) = (3, 7)$ in a corresponding process cycle. Thereby, in order to complete one execution cycle of *P3* in this case, it will read data from *ch3* which will be produced during the next execution cycle of *P2*. 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 Def. 13, 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}. In this work, we use the *Barvinok* library [17] to count integer points inside a parametric polytope. The *Barvinok* library can solve the counting problem in polynomial time. In general, the number of integer points inside a parametric polytope is defined as a list of (quasi-)polynomials. A quasi-polynomial is a polynomial with periodic numbers as coefficients. For instance, considering input port *IP3* shown in Fig. 4, $D_{IP3}^{CYC}(M, N) = \{(i, j) \in \mathbb{Z}^2 \mid 0 \leq i \leq M \wedge 0 \leq j \leq N - 2i\}$. The number of tokens $|D_{IP3}^{CYC}(M, N)|$ read by function `READ(in2, IP3)` in one process cycle is rep-

Algorithm 1: Generation of polynomials for function *Check*

Input: A P^{3}N
Result: A list of (quasi-)polynomials
1 **foreach** *channel* Ch *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* $q_{OP}(\vec{\mathbf{p}_j})$ *in* $|D_{OP}^{CYC}|$ **do**
4 Get chamber C ;
5 **foreach** *(quasi-)polynomial* $q_{IP}(\vec{\mathbf{p}_t})$ *in* $|D_{IP}^{CYC}|$ **do**
6 Get chamber C' ;
7 Compute $q_{res}(\vec{\mathbf{p}_{jt}}) = q_{OP}(\vec{\mathbf{p}_j}) - q_{IP}(\vec{\mathbf{p}_t})$;
8 Compute chamber $C_{res} = C \cup C'$;
9 **if** $q_{res}(\vec{\mathbf{p}_{jt}}) = 0$ **then**
10 Consistency is preserved for chamber C_{res};
11 **else if** $q_{res}(\vec{\mathbf{p}_{jt}})$ *is a non-zero constant* **then**
12 Eliminate chamber C_{res};
13 **else**
14 Store (quasi-)polynomial $q_{res}(\vec{\mathbf{p}_{jt}})$ with C_{res} ;

resented as the list of polynomials found by the *Barvinok* library:

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

where $C1 = \{(M,N) \in \mathbb{Z}^2 \mid M \leq N \wedge 2M \geq 1+N\}$ and $C2 = \{(M,N) \in \mathbb{Z}^2 \mid 2M \leq N\}$ are called *chambers*. Also, the second polynomial is a quasi-polynomial, in which $\{0,1\}_N$ is the 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 Def. 13. Algorithm 1 summarizes the analysis we performed at design-time. Recall that the condition $|D_{OP}^{CYC}| = |D_{IP}^{CYC}|$ must be satisfied for a consistent execution of a P^{3}N. Thus, for each channel connecting dataflow processes, we first compute $|D_{OP}^{CYC}|$ and $|D_{IP}^{CYC}|$ (line 2). 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 equivalent. We check the equivalence by subtracting the (quasi-)polynomials from both lists symbolically. The symbolic subtraction (line 7) can result in a zero constant, non-zero constant, or (quasi-)polynomial. If the result is a zero constant (line 9), 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 (line 11), 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 (line 14), 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 $q_{res}(\vec{\mathbf{p}_{jt}}) = 0$. 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 that match an entry in the table. In the second alternative, function *Check* evaluates $q_{res}(\vec{\mathbf{p}_{jt}})$ against zero with incoming parameter values at run-time.

Let us consider the example shown in Fig. 4 again. We apply Algorithm 1 to extract the valid parameter combinations. Besides $|D_{IP3}^{CYC}(M,N)|$ as given in Eq. 1, $|D_{OP3}^{CYC}(M)| = 3M+4$ is obtained. Subtraction between the (quasi-)polynomials in $|D_{OP2}^{CYC}(M)|$

```
[ 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)

```
[ 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 par_ok = true
 else return par_ok = false
}
```
(b)

Figure 5: Two alternatives of Function *Check* in Fig. 2(b).

and $|D_{IP2}^{CYC}(M,N)|$ yields two $q_{res}(M,N)$:

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

where chambers $C1$ and $C2$ are equal to the chambers in Eq. 1. Clearly this correspond to the third case in Algorithm 1 (line 14). The structure of two alternatives of function *Check* is given in Fig. 5. The solutions of Eq. 2 stored in table *tab* is shown in Fig. 5(a), whereas evaluating Eq. 2 directly against zero at run-time is depicted in Fig. 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 parameter combinations are 34, and if $0 \leq M, N \leq 10000$, the number of combinations is 114.

4. EXPERIMENTS AND 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 2 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 open-source ESPAM toolflow [11] and the Xilinx Platform Studio (XPS) tool. The generated MPSoCs consist of several MicroBlaze 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 MicroBlaze processor.

The P^{3}N we consider is depicted in Fig. 6. It is formed by the processes in Fig. 4 and one additional process *P4*. Fig. 6 also shows the representation of processes *P3* and *P4* in order to show the domains $D_{OP5}^{CYC}(M,N)$ and $D_{IP5}^{CYC}(N)$ of ports *OP5* and *IP5*, connected to channel *ch5*. Consequently, applying Algorithm 1 yields the following two polynomials for channel *ch5*:

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

where $C1 = \{(M,N) \in \mathbb{Z}^2 \mid M \leq N \wedge 2M \geq 1+N\}$ and $C2 = \{(M,N) \in \mathbb{Z}^2 \mid 2M \leq N\}$. For channel *ch3*, the dependence relation of parameters M and N is already given in Eq. 2.

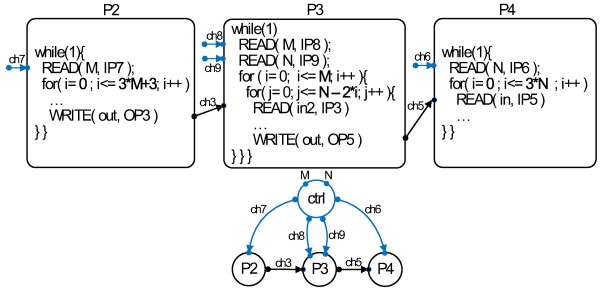

Figure 6: P^{3}N of our experiment

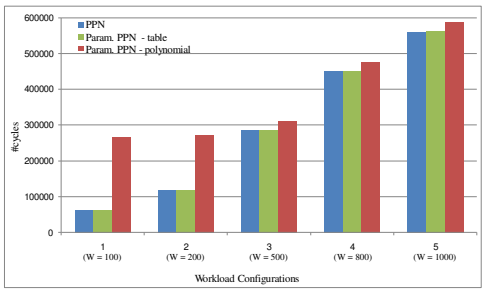

Figure 7: Performance results of PPN and P^{3}N implementations

In a first implementation alternative, we solved Eq. 2 and Eq. 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 Eq. 2 and Eq. 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 *P2*, *P3*, and *P4*. 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 $0 \leq M, N \leq 100$[1].

Fig. 7 shows the execution time of the reference PPNs and the two alternative P^{3}N implementations, respectively. The vertical axis in Fig. 7 corresponds to the execution time of the networks. The horizontal axis corresponds to different workload configurations. The workload W (in clk cycles) indicates execution latency of *P2*, *P3*, and *P4* in different workload configurations. In the all five workload configurations, the P^{3}Ns in which the checking for valid parameter combinations has been implemented with a table causes negligible run-time overhead, when compared to the reference PPNs (see the second bar and the first bar of each workload configuration, respectively).

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 execution latency of processes *P2*, *P3*, and *P4* is very small. That is, the dataflow processes are very light-weight, therefore, they are mostly blocked on reading from the control channels 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 of the dataflow processes, then the introduced overhead is significantly reduced, see configurations 3, 4, and 5 in Fig. 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 channels, 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 MicroBlaze 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.

[1]The evaluation results using parameter ranges, from $0 \leq M, N \leq 100$ to $0 \leq M, N \leq 10000$ and different number of reconfigurations are consistent but not reported here due to space limitations.

5. CONCLUSIONS

In the paper, we introduced the Parameterized Polyhedral Process Network (P^{3}N) model that is able to capture adaptive/dynamic application behavior. Such behavior is usually expressed by parameters which values are updated at run-time. We proposed a design-time approach which enables consistent execution of the P^{3}N model at run-time. We evaluated the possible run-time overhead caused by the parameterization of the P^{3}N model by designing and executing MPSoCs on an FPGA-based platform. The obtained results show that the parameterization we proposed is efficient in terms of the execution overhead introduced by the implementation of the process networks.

Acknowledgements

The research leading to these results has been partly supported by the ARTEMIS Joint Undertaking under grant agreement no. 100029, from SenterNovem, and Dutch Technology Foundation STW, project NEST, no. 10346. We also thank the anonymous reviewers for providing thoughtful comments.

6. REFERENCES

[1] A. Gerstlauer et al. Electronic System-Level Synthesis Methodologies. *IEEE TCAD*, 28(10):1517–1530, Oct. 2009.

[2] C. Bastoul. Code Generation in the Polyhedral Model Is Easier Than You Think. In *Proc. of PACT*, pages 7–16, 2004.

[3] B.D. Theelen et al. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. *In Proc. of MEMOCODE*, 0:185–194, 2006.

[4] B. Bhattacharya and S. Bhattacharyya. Parameterized dataflow modeling for DSP systems. *IEEE Trans. Signal Process.*, 2001.

[5] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static data flow. *IEEE Trans. Signal Process.*, 1996.

[6] J. T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. PhD thesis, EECS Department, University of California, Berkeley, 1993.

[7] E. A. de Kock. Multiprocessor mapping of process networks: a JPEG decoding case study. In *Proc. of ISSS*, pages 68–73, 2002.

[8] P. Feautrier. Automatic Parallelization in the Polytope Model. In *The Data Parallel Programming Model*, pages 79–103, 1996.

[9] P. Feautrier. Scalable and structured scheduling. *Int. J. Parallel Program.*, 34:459–487, October 2006.

[10] M. Geilen and T. Basten. Reactive process networks. In *Proc. of EMSOFT*, pages 137–146, New York, NY, USA, 2004. ACM.

[11] H. Nikolov et al. Systematic and automated multiprocessor system design, programming, and implementation. *IEEE TCAD*, 2008.

[12] G. Kahn. The semantics of a simple language for parallel programming. In *Proc. of Information Processing*. 1974.

[13] E. Lee and D. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 1987.

[14] L. Mandel, F. Plateau, and M. Pouzet. Lucy-n: a n-Synchronous Extension of Lustre. In *Proc. of MPC*, Québec, Canada, June 2010.

[15] S. Neuendorffer and E. Lee. Hierarchical reconfiguration of dataflow models. In *Proc. of MEMOCODE*, pages 179–188, June 2004.

[16] S. Meijer et al. Throughput modeling to evaluate process merging transformations in polyhedral process networks. In *DATE, 2010*.

[17] S. Verdoolaege et al. Counting integer points in parametric polytopes using barvinok's rational functions. *Algorithmica*, 2007.

[18] T. Stefanov et al. System design using kahn process networks: The compaan/laura approach. In *Proc. of DATE*, pages 340–345, 2004.

[19] S. Verdoolaege. *Handbook on signal processing systems*, chapter Polyhedral process networks. Springer, 2010.

[20] S. Verdoolaege, H. Nikolov, and T. Stefanov. pn: a tool for improved derivation of process networks. *EURASIP J. Embedded Syst.*, 2007.

[21] W. Haid et al. Efficient execution of kahn process networks on multi-processor systems using protothreads and windowed FIFOs. In *Proc. of ESTIMedia*, pages 35–44, Grenoble, France, 2009. IEEE.