Mapping of Streaming Applications Considering Alternative Application Specifications

JIALI TEDDY ZHAI, HRISTO NIKOLOV, and TODOR STEFANOV, Leiden University, The Netherlands

Streaming applications often require a parallel Model of Computation (MoC) to specify their application behavior and to facilitate mapping onto Multi-Processor System-on-Chip (MPSoC) platforms. Various performance requirements and resource budgets of embedded systems ask for an efficient design space exploration (DSE) approach to select the best design from a design space consisting of a large number of design choices. However, existing DSE approaches explore the design space that includes only architecture and mapping alternatives for an initial application specification given by the application designer. In this article, we first show that a design often might not be optimal if alternative specifications of a given application are not taken into account. We further argue that the best alternative specification consists of only independent and load-balanced application tasks. Based on the *Polyhedral Process Network* (PPN) MoC, we present an approach to analyze and transform an initial PPN to an alternative one that contains only independent processes if possible. Finally, by prototyping real-life applications on both FPGA-based MPSoCs and desktop multi-core platforms, we demonstrate that mapping the alternative application specification results in a large performance gain compared to those approaches, in which alternative application specifications are not taken into account.

Categories and Subject Descriptors: F.1.2 [**Computation by Abstract Devices**]: Model of Computation—*Parallelism and concurrency*; F.3.2 [**Logics and Meaning of Programming**]: Semantics of Programming Languages—*Process models*

General Terms: Design, Performance, Theory

Additional Key Words and Phrases: Streaming applications, alternative application specifications, polyhedral process networks, parallel programming, data-level parallelism, multi-processor system-on-chip

ACM Reference Format:
Zhai, J. T., Nikolov, H., and Stefanov, T. 2013. Mapping of streaming applications considering alternative application specifications. ACM Trans. Embedd. Comput. Syst. 12, 1, Article 34 (March 2013), 21 pages.
DOI = 10.1145/2435227.2435230 http://doi.acm.org/10.1145/2435227.2435230

1. INTRODUCTION

Streaming applications are widely used in the context of audio, video, and digital signal processing domains. Stringent performance requirements such as high throughput and/or low latency have been driving the implementation of streaming applications on Multi-Processor System-on-Chip (MPSoC) platforms, which contain an increasing number of Processing Elements (PEs).

The research leading to these results has been partly supported by Dutch Technology Foundation STW, project NEST, no. 10346.
Authors' addresses: J. T. Zhai, H. Nikolov, and T. Stefanov, Leiden Institute of Advanced Computer Science, Leiden University, Netherlands; email: {tzhai, nikolov, stefanov}@liacs.nl.

DOI 10.1145/2435227.2435230 http://doi.acm.org/10.1145/2435227.2435230

To facilitate the MPSoC design, application behavior is usually specified using a certain parallel *Model of Computation* (MoC), in which the application is modeled as several concurrently executing and communicating tasks. The task-level parallelism is thus naturally exposed. The MPSoC platforms are modeled using a set of PEs interconnected via certain communication infrastructure. Both application and platform models enable systematic, model-based approaches [Gerstlauer et al. 2009] for efficient MPSoC design. Among all steps in these design approaches, efficient mapping of application tasks onto MPSoC platform resources plays a vital role and obtaining such efficient mapping is very challenging. During the mapping step, platform resources such as PEs are allocated and assignment of application tasks to the platform resources is determined. In most of the cases, all possible combinations of PE allocation and assignment of application tasks to PEs constitute an enormous design space. To efficiently search the design space and find an optimum mapping solution, various Design Space Exploration (DSE) approaches proposed in the literature try to find a mapping that delivers the maximum achievable system performance for a given number of PEs. Such a mapping is called a *Pareto-optimal* point in the design space if higher performance cannot be achieved with fewer PEs. Currently, existing DSE approaches search the design space using different algorithms, e.g., stepwise refinement [Gerstlauer et al. 2008], heuristics [Stuijk et al. 2007], evolutionary algorithms [Pimentel et al. 2006; Thiele et al. 2007], branch-and-bound [Cong et al. 2009], and constraint programming [Zhu et al. 2010]. Existing DSE approaches consider only a single application specification given by application designers.

However, the given application specification may not be the most appropriate one for the considered MPSoC platform. The authors in [Kudlur and Mahlke 2008] showed that, for a set of representative streaming benchmarks, the theoretical speedup of mapping the initial parallel application specifications, given by the application designer, can only reach up to a limited number. This is because application designers mainly focus on realizing certain application behavior, including the identification of the functionality of application tasks and the synchronization/communication between these tasks. The computational capacity and communication cost of the MPSoC platform are often not taken into account when developing a parallel application specification. As a consequence, overwhelming communication between application tasks may cancel out the expected performance improvement when the application tasks are executed concurrently. Also, if the number of available PEs is greater than the number of application tasks in an initial specification, the aforementioned DSE approaches are incapable of exploring the mapping possibilities that utilize all PEs.

The discussion above indicates that alternative application specifications may be needed for efficient mapping of an application. In this article, we consider an alternative application specification as a different description of the same application behavior using the same MoC. For the same application behavior, there exists a large number of alternative specifications. Among them, the considered application specification should be the one that best matches the underlying MPSoC platform. In general, the best application specification, if it exists, to be mapped onto n PEs is the one that consists of n independent and load-balanced tasks. Then, without complex DSE, subsequently mapping these n tasks onto n PEs will always result in n times speedup, because all PEs are equally loaded and 100% utilized without the need to synchronize and communicate data with each other.

Therefore, in this article we study the problem of whether an alternative specification exists for an initial parallel application specification, which consists of only independent and load-balanced tasks. We solve the problem in the case when an application is initially modeled using the *Polyhedral Process Network* (PPN) MoC [Verdoolaege et al. 2007a]. Specifically, we divide the problem into two stages. In the first stage, given an

application initially modeled using the PPN MoC, we analytically identify independent execution of PPN processes,[1] called *communication-free partitions*. If they exist, the initial PPN is automatically transformed to a set of communication-free partitions, i.e., an alternative PPN. In the second stage, the application mapping problem is considered as grouping the set of obtained communication-free partitions to balance the application workloads across all PEs. To achieve the load-balancing, any existing DSE algorithm can be leveraged. As a result, mapping an application using this alternative PPN leads to better performance than mapping the initial PPN.

1.1. Motivating Example

To demonstrate the importance and usefulness of considering alternative application specifications, let us consider an example application modeled using the PPN MoC shown in Figure 1(a). It represents a common topology of a parallel application specification and consists of three PPN processes *P1*, *P2*, and *P3* communicating data via FIFO channels. Note that *P3* has cyclic data dependences through channel *E3*. The behavior of each PPN process is given as C code above the corresponding process. Besides the PPN processes expressing the application behavior, the nodes *src* and *snk* represent the environment which provides input data and collects results. The formal definition of the PPN MoC is given in Section 2.1. Suppose that nodes *src* and *snk* are much faster than the PPN processes and the PPN is to be mapped onto the platform shown in Figure 2(a). The type of platforms we assume is stated in Section 1.3. The workloads of functions *F1*, *F2*, and *F3* in Figure 1(a) on the PEs are 6, 100, and 30 time units, respectively. The communication latency via the interconnection structure is assumed to be 5 time units. Naturally, the maximum performance of mapping the initial PPN can be achieved if each PPN process is mapped onto a separate PE, namely 3 PEs in this example. In case that more than 3 PEs are available, the existing DSE approaches are incapable of exploring the mapping possibilities that utilize all PEs. Thus, further performance improvements of the system are not explored. In fact, considering only the initial PPN shown in Figure 1(a), only 2 PEs are required to achieve the maximum performance if we perform DSE to obtain pareto-optimal mappings of processes. That is, processes *P1* and *P2* are pipelined and mapped onto *PE1*, while process *P3* is mapped onto *PE2* as shown in Figure 2(a). Figure 2(b) shows the achieved speedup of pareto-optimal mappings of the initial PPN (denoted as *Initial*).

However, more parallelism is exposed and higher performance can be achieved, if the initial PPN is transformed to a set of communication-free partitions. A communication-free partition corresponds to a subset execution of PPN processes to produce an output of the PPN, without the need to communicate data with other partitions. To illustrate communication-free partitions, the execution of each PPN process in Figure 1(a) is visualized in Figure 1(b). The dots represent individual iterations of the PPN processes. For example, one iteration of *P3* comprises one execution of its loop body (lines 3–10 of *P3* in Figure 1(a)). The arrows between iterations denote data dependences. For this example, the initial PPN can be transformed to 8 communication-free partitions denoted as *Parti. 0–7* in Figure 1(b) (each partition is surrounded by a dashed box). One can see in Figure 1(b) that no arrows (data dependences) exist across the partitions. Each partition contains a subset execution of PPN processes *P1*, *P2*, and *P3* in Figure 1(a). After communication-free partitioning, the initial PPN in Figure 1(a) is transformed to the alternative PPN shown in Figure 3(a). The only communication between PEs occurs when input data is demultiplexed from node *src* to all partitions

[1]In this article, processes and tasks are used interchangeably.

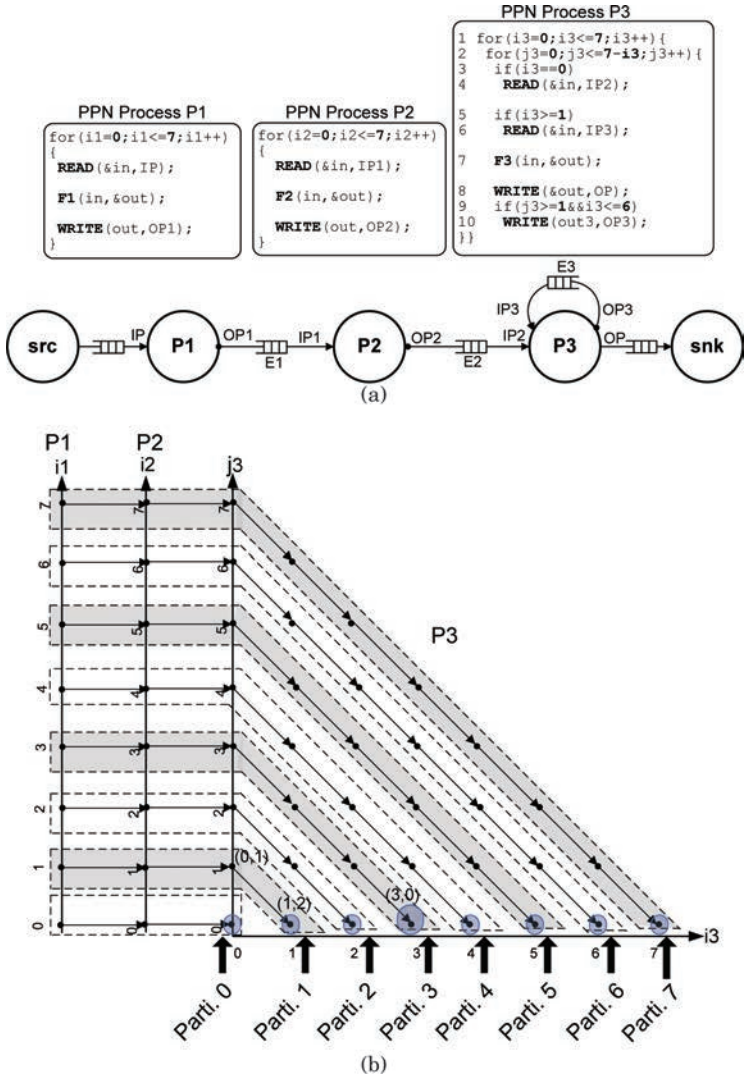


Fig. 1. (a) An example of a PPN; (b) execution of the PPN processes and its communication-free partitions.

and output produced by the partitions is multiplexed to node *snk*. For example, this can be seen with the help of Figure 1(b). In the initial PPN, node *src* sends the input data to *P1* at its iterations from (0) to (7) due to a dependence relation (the definition of dependence relations is given in Section 2.1). In the alternative PPN, with the same dependence relation, node *src* sends the input data at iteration (0) of *P1* to partition *Parti. 0*, the input data at iteration (1) of *P1* to partition *Parti. 1*, and so on. Analogously, in the alternative PPN, node *snk* collects the output data produced at iteration $(0, 0)$ of *P3* from partition *Parti. 0*, the output data produced at iterations $(0, 1)$ and $(1, 2)$ of *P3* from partition *Parti. 1*, and so on. With a given dependence relation in the initial PPN, the correct demultiplexing and multiplexing in the alternative PPN from the data source to all partitions and from all partitions to the data sink are automatically generated by our approach (see Section 2.4.2 for details). Except the communication

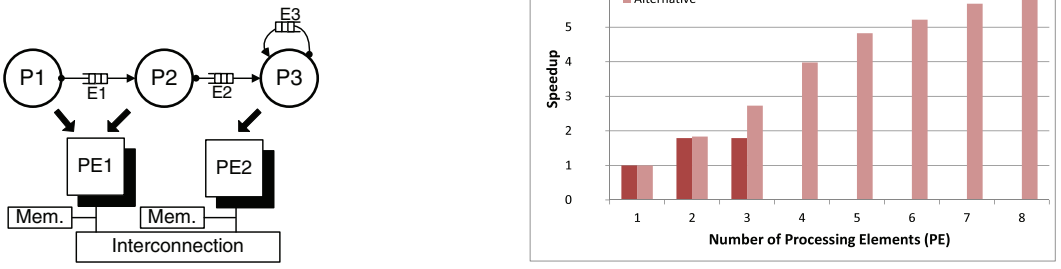


Fig. 2. (a) Mapping of the PPN in Figure 1(a) onto 2 PEs achieving the maximum performance; (b) performance results of mapping the initial PPN and the alternative PPN after communication-free partitioning.

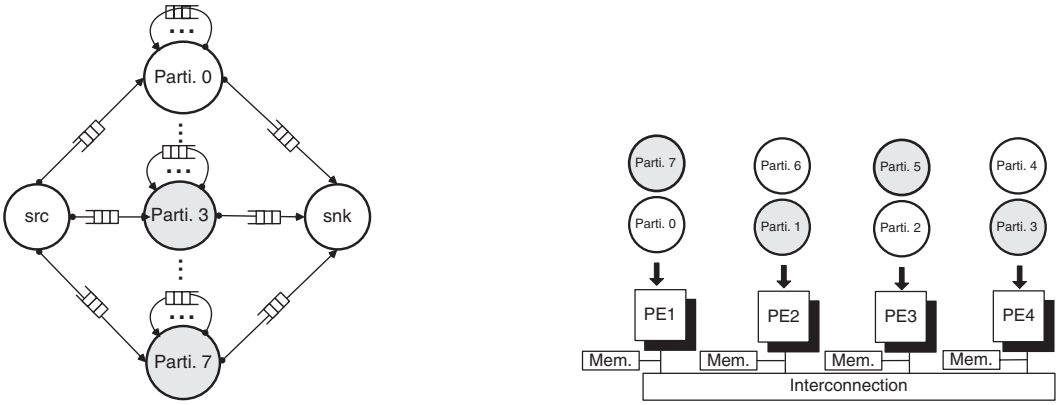


Fig. 3. (a) The alternative specification of the PPN in Figure 1(a) after communication-free partitioning; (b) mapping of the alternative specification onto 4 PEs (the data source and sink as well as all channels connected to both of them are omitted for succinctness).

between the partitions and the data source/sink, mapping the obtained partitions onto PEs will only result in local communication whose cost can be neglected on any platform. For instance, in case of 4 PEs available, mapping the derived alternative PPN in Figure 3(a) is shown in Figure 3(b).

Figure 2(b) also shows the achieved speedup of pareto-optimal mappings of the alternative PPN in Figure 3(a) (denoted as *Alternative*). Compared to mapping the initial PPN, mapping the alternative PPN constantly leads to a better performance. Moreover, the alternative PPN allows us to utilize up to 8 PEs, thereby achieving even higher speedup, which is not possible by considering only the initial PPN. Figure 2(b) shows that, for the alternative PPN, a linear speedup is observed up to 5 PEs. This is because the grouping of the 8 communication-free partitions can balance the workloads across up to 5 PEs. For instance, the 4 groups with 2 partitions each shown in Figure 3(b) have the same workload, i.e., the total number of iterations (dots) in all such 4 groups is equal. The speedup of mapping the derived alternative PPN onto 6 to 8 PEs saturates due to unbalanced workloads. From this motivating example, we can see the necessity and usefulness of considering alternative application specifications, particularly the one containing only communication-free and load-balanced partitions.

1.2. Research Contributions

For an application modeled using the *Polyhedral Process Network* (PPN) MoC [Verdoolaege et al. 2007a], we devise an approach to analytically determine the maximum amount of data-level parallelism, i.e., the number of communication-free partitions. Subsequently, we propose a procedure to transform the initial PPN to a set of communication-free partitions, if they exist. Our approach also can be applied to applications with cyclic dependences, which are traditionally considered as performance bottleneck and hard to parallelize. Finally, we demonstrate and validate the effectiveness of our approach by prototyping real-life streaming applications on FPGA-based MPSoCs and a desktop multi-core platform, each of which has different computational and communication characteristics.

1.3. Scope of Work

In this article, we consider streaming applications which can be modeled using the PPN MoC. To facilitate application modeling, the PPN MoC can be derived automatically using the pn compiler [Verdoolaege et al. 2007a] or *Polly* [Grosser et al. 2011], which accept a sequence of affine nested loop programs in C. We assume that there are only one data source and sink and they are orders of magnitude faster than the computational tasks of the applications. Furthermore, the achievable performance of a PPN is not constrained by the buffer size required for each communication channel. It is possible to compute a buffer size for each PPN channel using the pn compiler such that larger buffer sizes do not increase the performance. We statically allocate a FIFO buffer for each PPN channel on target platforms. The target platforms considered in this article are homogeneous MPSoCs consisting of programmable PEs interconnected via any type of communication infrastructure. After communication-free partitioning, we assume that one partition completely fits onto one PE, in terms of program and data memory usage.

1.4. Related Work

An alternative application specification modeled as a Synchronous Dataflow (SDF) [Lee and Messerschmitt 1987] MoC is considered in Yang and Ha [2009]. All actors in the initial SDF graph are completely unfolded to their equivalent Homogeneous SDF actors (production/consumption rate equal to 1). The unfolding leads to an exponential increase in the size of the graph. Therefore, the authors propose a heuristic based on an evolutionary algorithm to find a mapping and a schedule for the resulting Homogeneous SDF graph. Compared to Yang and Ha [2009], we consider a more expressive model than SDF, i.e., the PPN MoC. Also, instead of completely unfolding all PPN processes (equal to unfolding actors in Yang and Ha [2009]), we operate on a compact representation to avoid the explosion in the size of the graph. Moreover, this compact representation also allows us to analytically determine the maximum amount of data-level parallelism to be exploited, i.e., the maximal number of communication-free partitions.

SDF is also used as the underlying MoC in Gordon et al. [2006] and Kudlur and Mahlke [2008]. Each SDF actor is assumed to have only one input and one output port. Based on this assumption, *stateless* actors (the actors without cyclic dependences) in the SDF graph are first fused into compound actors. Then, those compound actors are duplicated by inserting *splitters* and *joiners* to distribute data and collect results. Compared to Gordon et al. [2006] and Kudlur and Mahlke [2008], the PPN MoC considered in this article is more general with an arbitrary number of input and output ports of PPN processes. The problem addressed in this article is thus more difficult as simple fusion-duplication is not applicable to PPN processes. Also, in Gordon et al. [2006] and

Kudlur and Mahlke [2008], *stateful* actors (see for instance process *P3* in Figure 1(a)) cannot be fused and duplicated. Instead, software pipelining techniques are applied to the stateful actors. It is based on the assumption that communication latency between different PEs (different pipeline stages) could be overlapped by computation. However, we believe, depending on the platform in use, the communication latency may not be hidden and completely overlapped by computation. In contrast, our approach tries to extract data-level parallelism even for the PPN processes with cyclic data dependences while completely avoiding communication between PEs.

In Liao et al. [2006], affine partitioning is used in the Brook language to map streaming applications. Similar to the affine partitioning, our communication-free partitioning also aims at obtaining coarse-grained PPN processes. In contrast, our partitioning strategy is able to completely eliminate communication, which might not be possible in some cases using affine partitioning.

The PPN MoC is used in Meijer et al. [2010]. The authors suggest that a perfect alternative application specification can be achieved by first partitioning PPN processes and then merging some PPN processes into a compound one. However, a procedure of partitioning and merging PPN processes is not discussed. In this article, we propose a systematic procedure to partition and merge PPN processes in a PPN.

2. SOLUTION APPROACH

We first introduce the application model used in this article, i.e., the PPN MoC, in Section 2.1 to better understand our contributions. Based on the PPN MoC, we translate the problem of finding communication-free partitions in a PPN to first finding all direct and indirect data dependences. The procedure of finding all data dependences in a PPN is presented in Section 2.2. In Section 2.3, we present an analytical framework to determine the number of communication-free partitions that can be derived from a PPN, for which all direct and indirect data dependencies are computed. We propose an algorithm in Section 2.4 to transform the PPN to such a set of communication-free partitions, if they exist.

2.1. Preliminaries - the PPN MoC

The PPN MoC [Verdoolaege et al. 2007a] represents a streaming application as a directed, multi-dimensional dataflow graph $G = (\mathcal{P}, \mathcal{E})$ which is a special case of the Kahn Process Network [Kahn 1974] MoC. That is, a PPN consists of several autonomously executing processes $P \in \mathcal{P}$ only communicating via FIFO channels $E \in \mathcal{E}$. PPN processes are synchronized through FIFOs, i.e. any process is blocked when attempting to read from an empty FIFO or write to a full FIFO. The execution of a PPN process is specified using affine nested *for*-loops, called *domain*. The domain can be represented in the polytope model [Feautrier 1996]. Formally, a domain D is defined as:

$$D = \{\vec{I} \in \mathbb{Z}^d \mid A \cdot \vec{I} \geq \vec{b}\},$$

where $A \in \mathbb{Z}^{m \times d}$, $\vec{b} \in \mathbb{Z}^d$, $\vec{I}$ is an *iteration vector*, and d indicates the nested-loop depth. At each iteration $\vec{I}$ during the execution of a PPN process P, namely $\vec{I} \in D_P$, P first reads data from input ports (*IP*) in the input port domain D_{IP} if $\vec{I} \in D_{IP}$. Then the process executes the process function (computation) and subsequently writes results to output ports (*OP*) in the output port domain D_{OP} if $\vec{I} \in D_{OP}$. The set of iterations, at which a PPN process writes data to the environment, are called *sink iterations*, denoted by D_{snk}. Furthermore, an affine function called *dependence relation* R_E is defined for each channel E, after performing *Array Dataflow Analysis* (ADA) [Feautrier 1991] in

Table I. Notations

D	domain (polytope)
$\|D\|$	cardinality of domain D
E	channel in a PPN
$\cap/\cup$	intersection/union of domains
n	the number of communication-free partitions
P	PPN process
R_E	affine dependence relation R associated with channel E
R^+	transitive closure of relation R
$R(\vec{I})$	slicing of relation R by a constant $\vec{I}$
$R(D)$	apply a domain D to relation R
ran$R$	range of relation R
dom$R$	domain of relation R

the pn compiler [Verdoolaege et al. 2007a]. It is formally specified as:

$$R_E = \{\vec{I} \rightarrow \vec{J} \in \mathbb{Z}^{d1} \times \mathbb{Z}^{d2} \mid \vec{I} \in D_{IP} \wedge \vec{J} \in D_{OP} \wedge \vec{J} = B \cdot \vec{I} + \vec{c}\},$$

where $B \in \mathbb{Z}^{d2 \times d1}$, $\vec{c} \in \mathbb{Z}^{d2}$. It indicates that data produced at iteration $\vec{J} \in D_{OP}$ is consumed at iteration $\vec{I} \in D_{IP}$ if output port OP is connected to input port IP via channel E. Finally, for the sake of convenience, the set of operators and notations used in this article is summarized in Table I.

Consider the PPN shown in Figure 1(a). The PPN reads data from the data source through input port IP and writes results to the data sink through output port OP. For instance, the sink iterations (see lines 1, 2, and 8 in Figure 1(a)) are represented as:

$$\begin{aligned} D_{snk} &= \left\{ (i3, j3) \in \mathbb{Z}^2 \mid \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ -1 & -1 \end{bmatrix} \cdot \begin{bmatrix} i3 \\ j3 \end{bmatrix} \geq \begin{bmatrix} 0 \\ -7 \\ 0 \\ -7 \end{bmatrix} \right\}, \\ &= \{(i3, j3) \in \mathbb{Z}^2 \mid 0 \leq i3 \leq 7 \wedge 0 \leq j3 \leq 7 - i3\}. \end{aligned} \tag{1}$$

For process $P3$, its execution is described by domain $D_{P3} = D_{snk}$ (see lines 1–2 of process $P3$ in Figure 1(a)). The iterations in process domain D_{P3} are illustrated as dots in Figure 4. Process $P3$ reads data from input port $IP3$ (see line 1–2 and 5 of process $P3$ in Figure 1(a)) in domain $D_{IP3} \subset D_{P3}$, denoted as:

$$D_{IP3} = \{(i3, j3) \in \mathbb{Z}^2 \mid 1 \leq i3 \leq 7 \wedge 0 \leq j3 \leq 7 - i3\}.$$

The iterations in input port domain D_{IP3} are surrounded by the solid triangle in Figure 4. After reading data from an input port $IP2$ or $IP3$ to initialize variable in, $P3$ performs computation by executing function F3. Finally, $P3$ writes data to output ports OP and $OP3$ (see lines 1–2 and 9 in Figure 1(a)) if the iteration $\vec{I} = (i3, j3)$ is in domain D_{OP3}, denoted as:

$$D_{OP3} = \{(i3, j3) \in \mathbb{Z}^2 \mid 0 \leq i3 \leq 6 \wedge 1 \leq j3 \leq 7 - i3\}.$$

The iterations in output port domain D_{OP3} are surrounded by the dotted triangle in Figure 4. In Figure 1(a), the dependence relation R_{E3} for channel $E3$, connecting output

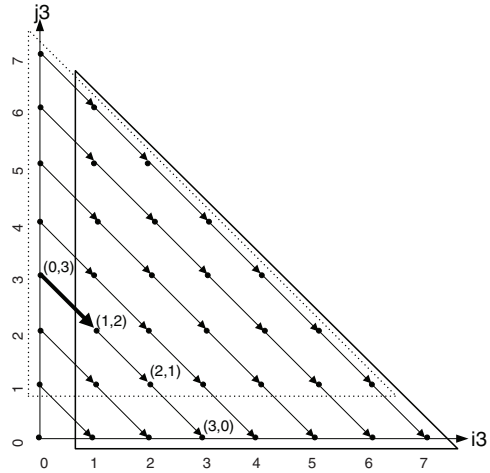


Fig. 4. Domain of PPN process *P3* in Figure 1(a), the input port domain of *IP3* (surrounded by the solid triangle), output port domain of *OP3* (surrounded by the dotted triangle), and dependence relation R_{E3} (denoted by the arrows between dots).

port *OP3* to input port *IP3*, is derived by performing ADA and denoted as:

$$\begin{aligned} R_{E3} = \Bigg\{ & (i3, j3) \rightarrow (i3', j3') \mid (i3, j3) \in D_{IP3} \wedge (i3', j3') \in D_{OP3} \wedge \begin{bmatrix} i3' \\ j3' \end{bmatrix} \\ & = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} i3 \\ j3 \end{bmatrix} + \begin{bmatrix} -1 \\ 1 \end{bmatrix} \Bigg\} \\ & = \{(i3, j3) \rightarrow (i3', j3') \mid (i3, j3) \in D_{IP3} \wedge (i3', j3') \in D_{OP3} \wedge i3' \\ & = i3 - 1 \wedge j3' = j3 + 1\}. \end{aligned} \tag{2}$$

R_{E3} is illustrated as the solid arrows in Figure 4. For example, the bold arrow shows that iteration $(1, 2)$ of process *P3* depends on iteration $(0, 3)$ of itself, denoted by $R_{E3} = \{(1, 2) \rightarrow (0, 3)\}$. The domain of R_{E3} is denoted by $\mathrm{dom}R_{E3} = D_{IP3}$ and the range of R_{E3} is denoted by $\mathrm{ran}R_{E3} = D_{OP3}$

2.2. Finding All Dependences in a PPN

For streaming applications, input data is read from the data source, subsequently processed by PPN processes at their iterations during the execution, and finally written to the data sink. Therefore, to generate an output of a PPN, i.e., the output produced at an iteration $\vec{I} \in D_{snk}$, it directly or indirectly depends on several iterations of PPN processes. To find out communication-free partitions in a PPN, we need to solve the problem of finding all "direct" and "indirect" data dependences in a PPN.

The direct dependences result immediately from the dependence relations. For example, dependence relation $R_{E3} = \{(1, 2) \rightarrow (0, 3)\}$ in Figure 4 (the bold arrow) expresses a direct dependence. In contrast, iteration $(2, 1)$ indirectly depends on iteration $(0, 3)$ through iteration $(1, 2)$. In this article, we formulate the problem of finding all direct and indirect data dependences by computing *transitive closure* [Kelly et al. 1996; Pugh and Rosser 1997], denoted by R^+, of affine dependence relation R. It is formally defined as:

$$\vec{I} \rightarrow \vec{J} \in R^+ \Leftrightarrow (\vec{I} \rightarrow \vec{J}) \in R \vee \exists \vec{K} \text{ s.t. } (\vec{I} \rightarrow \vec{K}) \in R \wedge (\vec{K} \rightarrow \vec{J}) \in R^+. \tag{3}$$

Now, as an example, considering the affine dependence relation R_{E3}, illustrated in Figure 1(b) (solid arrows), and its transitive closure R^+_{E33}, we can have for instance an indirect dependence:

$$(2,1) \rightarrow (0,3) \in R^+_{E33} \Leftrightarrow \exists (1,2) \text{ s.t. } (2,1) \rightarrow (1,2) \in R_{E3} \wedge (1,2) \rightarrow (0,3) \in R_{E3}.$$

From Equation (3), we can see that "direct" and "indirect" dependences are uniformly expressed as transitive closure of dependence relations. Thus, we use the term *transitive dependences* to denote both types of dependences. Note that transitive closure of a set of affine relations is not an affine form in general. An under-approximated and closed affine form is computed in Kelly et al. [1996]. In contrast, we consider an affine over-approximation in case of non-affine closed form. The over-approximation first guarantees that a valid schedule always can be found for each communication free partition at the cost of potentially fewer communication-free partitions. Second, existing powerful code generation methods for affine dependence relations still can be leveraged.

Now, finding all transitive dependences in a PPN is translated to computing transitive closure of all dependence relations. Therefore, we first take a union R_{deps} of all dependence relations in a PPN as:

$$R_{deps} = \bigcup_{\forall E \in \mathcal{E}} R_E.$$

Subsequently, we can compute the transitive closure of the union R_{deps}. In this article, we use the *isl* library [Verdoolaege 2010] to compute the transitive closure of affine dependence relations in a potentially over-approximated closed form. For the PPN in Figure 1(a), computing the union of all dependence relations yields:

$$R_{deps} = R_{E1} \cup R_{E2} \cup R_{E3}.$$

Then, by computing the transitive closure of R_{deps}, we obtain:

$$R^+_{deps} = R_{deps} \cup R^+_{E13} \cup R^+_{E23} \cup R^+_{E33},$$

where R^+_{E13}, R^+_{E23}, and R^+_{E33} are transitive dependence relations, represented as follows:

$$R^+_{E13} = \{(i3, j3) \rightarrow (i1) \mid 0 \leq i3 \leq i1 \wedge i1 \leq 7 \wedge i1 = i3 + j3\}, \tag{4a}$$

$$R^+_{E23} = \{(i3, j3) \rightarrow (i2) \mid 0 \leq i3 \leq i2 \wedge i2 \leq 7 \wedge i2 = i3 + j3\}, \tag{4b}$$

$$R^+_{E33} = \{(i3, j3) \rightarrow (i3', j3') \mid 1 \leq i3 \leq 7 \wedge 0 \leq j3 \leq 7 - i3 \wedge 0 \leq i3' \leq 6 \wedge 0 \leq i3' \leq i3 + j3 - 1 \wedge j3' = i3 + j3 - i3'\}. \tag{4c}$$

After computing the transitive closure of all dependence relations in the PPN in Figure 1(a), 3 extra channels *E13*, *E23*, and *E33* corresponding to the transitive dependence relations are added in the PPN as shown in Figure 5(a). For the execution of the PPN (domains of PPN processes *P1*, *P2*, and *P3*) shown in Figure 1(b), a set of transitive dependences is illustrated as dashed arrows in Figure 5(b). For instance, $R_{E33^+} = \{(3,0) \rightarrow (0,3)\}$, shown as the bold and dashed arrow, indicates that iteration $(3,0)$ of PPN process *P3* transitively depends on iteration $(0,3)$ of itself.

2.3. Computing the Number of Communication-Free Partitions

As explained in Section 2.2, we derive all dependent iterations that generate an output at any iteration $\vec{I} \in D_{snk}$. Based on this information, in this section, we compute the number of communication-free partitions that can be derived from a given PPN.

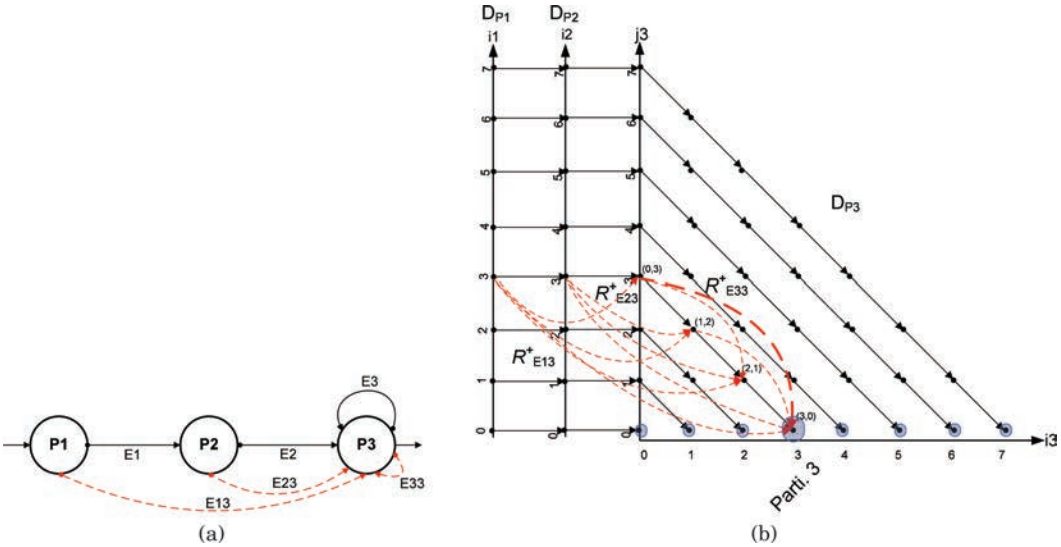


Fig. 5. (a) Transitive dependences of the PPN in Figure 1(a); (b) the set of transitive dependences for communication-free partition *Parti. 3* in Figure 1(b)

Essentially, we need to find a set of iterations in domain D_{snk} that are independent from each other. Each of these iterations identifies a distinct communication-free partition (see the dashed boxes in Figure 1(b)). Consider the PPN in Figure 1(a) and its execution illustrated in Figure 1(b). As explained in Section 2.1, $D_{snk} = D_{P3}$ (see the triangular part in Figure 1(b) denoted as *P3*). Our goal is to find the 8 iterations marked by circles in Figure 1(b). It can be seen that they are independent of each other and they identify the 8 communication-free partitions. Therefore, the number of these iterations determines the number of communication-free partitions.

In general, to find the set of iterations mentioned above, we first state the following lemma.

LEMMA 2.1. *Any transitive dependence relation R_E^+ is a total and surjective affine relation, which maps iterations $\vec{I}$ in input port domain D_{IP} to iterations $\vec{J}$ in output port domain D_{OP}.*

PROOF. Totality of a transitive dependence relation R_E^+ holds because of the following property of the PPN MoC. For streaming applications operating on infinite input streams, we are only interested in consistent and deadlock-free PPNs. Therefore, if an iteration in an input domain ($\vec{I} \in D_{IP}$) is unmapped, it means that the PPN will deadlock at this iteration during execution of the PPN. At the same time, R_E^+ is also surjective, because several iterations in an input domain can be mapped to the same iteration in an output domain. This can be seen from the definition of the transitive closure of affine relations in Equation (3). If there exists $\vec{I} \rightarrow \vec{K} \in R$ and $\vec{K} \rightarrow \vec{J} \in R$, both iterations $\vec{I}$ and $\vec{K}$ are mapped to iteration $\vec{J}$ in R^+. □

For instance, transitive relation R_{E23}^+ in Figure 5(b) denotes that iterations $(0, 3)$, $(1, 2)$, $(2, 1)$, and $(3, 0)$ of *P3* are mapped to iteration (3) of *P2*.

Based on Lemma 2.1, we can have the following theorem.

THEOREM 2.1. *For any PPN, the number n of communication-free partitions is computed as* $n = |D^{ind}_{snk}|$, *where* $D^{ind}_{snk} \subseteq D_{snk}$ *and*

$$\begin{aligned} D^{ind}_{snk} = \; & \{\vec{I} \in \mathbb{Z}^d \mid \exists R^+ : \vec{I} \to \vec{J} \in R^+ \wedge \vec{I} \in D_{snk} \wedge \vec{J} \in D_{snk} \wedge \vec{I} \in (\mathrm{dom}R^+ - \mathrm{ran}R^+)\} \\ & \bigcup \{\vec{I} \in \mathbb{Z}^d \mid \forall R^+ : \vec{I} \to \vec{J} \in R^+ \wedge \vec{I} \in D_{snk} \wedge \vec{J} \notin D_{snk} \wedge \vec{I} \in \mathrm{dom}R^+\}. \end{aligned} \tag{5}$$

PROOF. For an iteration $\vec{I} \in D_{snk}$, it satisfies one of two mutually exclusive conditions. That is, the iteration either transitively depends on other iterations $\vec{J} \in D_{snk}$, or does not transitively depend on any iteration $\vec{J} \in D_{snk}$. The former condition is stated as $\vec{I} \to \vec{J} \in R^+ \wedge \vec{J} \in D_{snk}$ in Equation (5), whereas the latter condition is expressed as $\vec{I} \to \vec{J} \in R^+ \wedge \vec{J} \notin D_{snk}$. For the former condition, the surjective property of a transitive dependence R^+ stated in Lemma 2.1 indicates that multiple iterations $\vec{I} \in \mathrm{dom}R^+ \subset D_{snk}$ may depend on the same $\vec{J} \in \mathrm{ran}R^+ \subset D_{snk}$. We thus need to find out distinct iterations $\vec{I} \in \mathrm{dom}R^+$, which are not mapped from any other iterations $\vec{I} \in D_{snk}$. It is essentially equivalent to computing the lexicographically maximal iteration $\vec{I}$ if $\vec{I} \to \vec{J} \in R^+$. Such iterations $\vec{I}$ can be found by $\mathrm{dom}R^+ - \mathrm{ran}R^+$. On the other hand, if an iteration $\vec{I} \in D_{snk}$ does not transitively depend on any other iteration $\vec{J} \in D_{snk}$, where $\vec{I} \neq \vec{J}$, all these iterations are independent. This means all such iterations can definitely find independent communication-free partitions. Finally all those iterations in domain $D^{ind}_{snk} \subseteq D_{snk}$ can be computed by taking the union as given in Equation (5). □

Consider the PPN in Figure 1(a). As explained in Section 2.1, the sink iterations D_{snk} are described in Equation (1). Upon computing transitive closure R^+_{deps} of all dependence relations presented in Section 2.2, there are three transitive dependence relations on D_{snk}, namely R^+_{E13}, R^+_{E23}, and R^+_{E33}. Among them, R^+_{E33} satisfies the condition $\vec{I} \to \vec{J} \in R^+ \wedge \vec{I} \in D_{snk} \wedge \vec{J} \in D_{snk}$ as stated in Equation (5). The domain and range of R^+_{E33} are:

$$\begin{aligned} \mathrm{dom}R^+_{E33} &= \{(i3, j3) \in \mathbb{Z}^2 \mid 1 \leq i3 \leq 7 \wedge 0 \leq j3 \leq 7 - i3\}, \\ \mathrm{ran}R^+_{E33} &= \{(i3, j3) \in \mathbb{Z}^2 \mid 0 \leq i3 \leq 7 \wedge 1 \leq j3 \leq 7 - i3\}. \end{aligned}$$

Then, $\mathrm{dom}R^+_{E33} - \mathrm{ran}R^+_{E33}$ in Equation (5) yields:

$$D^{ind1}_{snk} = \{(i3, j3) \in \mathbb{Z}^2 \mid 1 \leq i3 \leq 7 \wedge j3 = 0\}. \tag{6}$$

Furthermore, we compute those iterations that satisfy the latter condition in Equation (5), namely they do not depend on any other iterations in domain D_{snk}. That is:

$$D^{ind2}_{snk} = \{(i3, j3) \mid i3 = 0 \wedge j3 = 0\}. \tag{7}$$

Finally, D^{ind}_{snk} can be computed by taking the union of D^{ind1}_{snk} obtained in Equation (6) and D^{ind2}_{snk} obtained in Equation (7):

$$\begin{aligned} D^{ind}_{snk} &= D^{ind1}_{snk} \cup D^{ind2}_{snk} \\ &= \{(i3, i3) \in \mathbb{Z}^2 \mid 0 \leq i3 \leq 7 \wedge j3 = 0\}. \end{aligned} \tag{8}$$

In general, D^{ind}_{snk} computed in accordance with Equation (5) is a union of domains represented by polytopes. Then, computing the number of communication-free partitions is equal to counting the number of integer points in the union of polytopes, denoted by $|D^{ind}_{snk}|$. The counting problem can be efficiently solved in polynomial time using the *barvinok* [Verdoolaege et al. 2007b] library. Finally, for the PPN shown in Figure 1(a) and D^{ind}_{snk} obtained in Equation (8), counting the number of integer points

in D^{ind}_{snk} yields $n = |D^{ind}_{snk}| = 8$. This confirms the same number of communication-free partitions, namely 8 as shown in Figure 1(b). Also, D^{ind}_{snk} corresponds to the iterations marked by circles show in both Figures 1(b) and 5(b).

2.4. Communication-Free Partitioning Algorithm

If the number of communication-free partitions computed in Section 2.3 is greater than 1, we can transform the initial PPN to a set of communication-free partitions. In Section 2.4.1, we first show an example of constructing one of the communication-free partitions for the PPN in Figure 1(a). Subsequently, we present the general partitioning algorithm in Section 2.4.2.

2.4.1. An Illustrative Example. Consider the PPN in Figure 1(a) and its execution illustrated in Figure 1(b). Let us for example assume that communication-free partition *Parti. 3* in Figure 1(b) is to be constructed. In the partitioning algorithm, our goal is to partition the domains of the PPN processes and obtain all iterations surrounded by the dashed box for *Parti. 3*. These iterations are transitively dependent on the iteration that identifies *Parti. 3*. In this case, *Parti. 3* is identified by iteration $(i3, j3) = (3, 0) \in D^{ind}_{snk}$ of process *P3* as computed in Equation (8). All transitive dependence relations R^+_{E33}, R^+_{E23}, and R^+_{E13} on iteration $(3, 0)$ are computed in Equations (4a) to (4c) and illustrated in Figure 5(b). In the first step of the partitioning algorithm for *Parti. 3*, we instantiate process instance $P3_{inst}$ (see Figure 6) of PPN process *P3* through R^+_{E33}. A process instance P_{inst} performs the same computational function as the original PPN process P does. The only difference is that the process instance P_{inst} only executes in a subdomain $D_{P_{inst}}$ of the original domain D_P. For *Parti. 3*, besides that iteration $(3, 0)$ belongs to domain $D_{P3_{inst}}$ of process instance $P3_{inst}$, $D_{P3_{inst}}$ contains also iterations $(2, 1)$, $(1, 2)$, and $(0, 3)$ of *P3*, on which iteration $(3, 0)$ depends, as shown in Figure 5(b). These iterations can be derived by "substituting" iteration $(3, 0)$ in R^+_{E33} (see Equation (4c)), denoted as $R^+_{E33}((3, 0))$:

$$\begin{aligned} R^+_{E33}((3,0)) &= \left\{(i3', j3') \mid (3,0) \rightarrow (i3', j3') \in R^+_{E33}\right\} \\ &= \{(i3', j3') \mid 0 \leq i3' \leq 2 \wedge j3' = 3 - i3'\}. \end{aligned} \tag{9}$$

Then, domain $D_{P3_{inst}}$ for *Parti. 3* can be obtained by taking a union of iteration $(3, 0)$ with the ones computed in Equation (9):

$$\begin{aligned} D_{P3_{inst}} &= (3,0) \cup R^+_{E33}((3,0)) \\ &= \{(i3', j3') \mid 0 \leq i3' \leq 3 \wedge j3' = 3 - i3'\}. \end{aligned} \tag{10}$$

Second, a process instance $P2_{inst}$ (see Figure 6) of PPN process *P2* is instantiated due to R^+_{E23} for *Parti. 3*. Domain $D_{P2_{inst}}$ contains iteration (3) of *P2* as shown Figure 5(b). It can be derived by "substituting" domain $D_{P3_{inst}}$, obtained in Equation (10), in R^+_{E23} (see Equation (4b)), denoted as $R^+_{E23}(D_{P3_{inst}})$:

$$\begin{aligned} D_{P2_{inst}} = R^+_{E23}(D_{P3_{inst}}) &= \{(j2) \mid (i3, j3) \rightarrow (j2) \in R^+_{E23} \wedge (i3, j3) \in D_{P3_{inst}}\} \\ &= \{(i2) \in \mathbb{Z} \mid i2 = 3\}. \end{aligned} \tag{11}$$

Finally, we need to instantiate a process instance $P1_{inst}$ (see Figure 6) with domain $D_{P1_{inst}}$ due to R^+_{E13}. Domain $D_{P1_{inst}}$ corresponds to iteration (3) in domain D_{P1} as shown in Figure 5(b). Analogous to obtaining domain $D_{P2_{inst}}$, domain $D_{P1_{inst}}$ can be obtained by "substituting" domain $D_{P3_{inst}}$ in R^+_{E13} (see Equation (4a)):

$$D_{P1_{inst}} = R^+_{E13}(D_{P3_{inst}}) = \{(i1) \in \mathbb{Z} \mid i1 = 3\}.$$

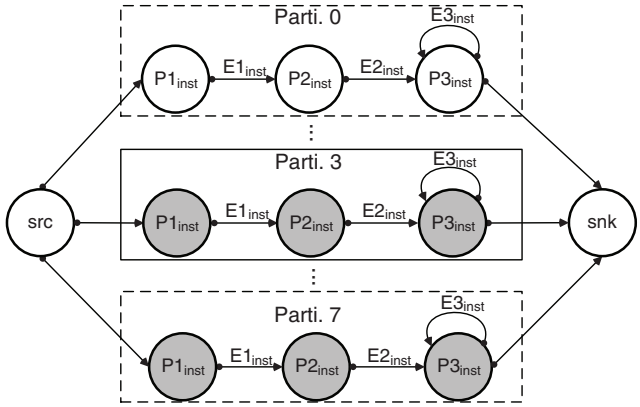


Fig. 6. The PPN in Figure 1(a) after communication-free partitioning.

Once all process instances for *Parti 3* are instantiated, next we instantiate channels for the process instances. Basically, if a channel in the initial PPN is incident with the process instances, a new channel is instantiated. For *Parti. 3*, channel *E3* in the initial PPN is incident with the process instance $P3_{inst}$. Then, a new channel $E3_{inst}$ is instantiated with the associated input port domain D_{IP3_inst}, output port domain D_{OP3_inst}, and dependence relation $R_{E3_{inst}}$:

$$\begin{aligned} D_{IP3\_inst} &= D_{IP3} \cap D_{P3_{inst}} \\ &= \{(i3, j3) \mid 1 \le i3 \le 3 \wedge j3 = 3 - i3\}, \\ D_{OP3\_inst} &= D_{OP3} \cap D_{P3_{inst}} \\ &= \{(i3', j3') \mid 0 \le i3' \le 2 \wedge j3' = 3 - i3'\}, \\ R_{E3_{inst}} &= \{(i3, j3) \rightarrow (i3', j3') \mid (i3, j3) \in D_{IP3\_inst} \wedge (i3', j3') \in D_{OP3\_inst} \\ &\quad \wedge i3' = i3 - 1 \wedge j3' = j3 + 1\}. \end{aligned} \tag{12}$$

Two other channels $E1_{inst}$, $E2_{inst}$ can be instantiated in a similar way, due to channels *E1*, *E2* in the initial PPN. In this way, communication-free partition *Parti. 3* shown in Figure 1(b) is constructed and illustrated by the solid box in Figure 6. In the next step, we merge all process instances $P3_{inst}$, $P2_{inst}$, and $P1_{inst}$ into a single compound process *Parti. 3* as shown in Figure 3(a). We generate a static schedule, similar to the one proposed in Verdoolaege et al. [2003], that executes all dependent iterations of the processes instances as soon as possible.

2.4.2. General Partitioning Algorithm. In general, to instantiate process instances and channels, we devise Algorithm 1 presented in this article. The input to Algorithm 1 is a PPN with all transitive dependences ($\mathcal{E}^+$) computed in Section 2.2 and $D^{ind}_{snk} \subseteq D_{snk}$ obtained in Theorem 2.1. Every sink iteration $\vec{K} \in D^{ind}_{snk}$ is used to identify a distinct communication-free partition. The output of Algorithm 1 is n communication-free partitions. The core part of the algorithm is presented below.

Algorithm 1 starts partitioning a PPN from the sink process, namely partitioning P_{snk} into n process instances P_{snk_inst}. For each iteration $\vec{K} \in D^{ind}_{snk}$ of the sink process, we instantiate a new process instance P_{snk_inst} (line 4). The loop (lines 3–17) iterates over all PPN processes to instantiate all process instances in all communication-free partitions. Basically, for a particular partition, we construct the domain for each process instance through all transitive dependence relations R^+_E on iteration $\vec{K}$. First, we construct

ALGORITHM 1: Communication-free partitioning procedure

Input: A $PPN = \{\mathcal{P}, \mathcal{E}\}$, $\mathcal{E}^+$, and D^{ind}_{snk} obtained in Theorem 2.1 ($n = |D^{ind}_{snk}|$).
Result: A $PPN' = \{\mathcal{P}', \mathcal{E}'\}$.

```
1  P' ← ∅, E' ← ∅ ;
2  Get sink process P_snk, D_{P_snk_inst} ← ∅ ;
3  foreach K⃗ ∈ D^ind_snk do
4      P_snk_inst ← P_snk ;
5      foreach Channel E+ ∈ E+ incident with P_snk do
6          Get R+_E associated with channel E+ ;
7          if K⃗ ∉ domR+_E then
8              continue;
9          if ranR+_E ⊆ D_snk then /* K⃗ depends on other iterations in D_snk */
10             D_{P_snk_inst} ← D_{P_snk_inst} ∪ K⃗ ∪ R+_E(K⃗) ;
11         else /* K⃗ depends on another process P */
12             D_{P_snk_inst} ← D_{P_snk_inst} ∪ K⃗ ;
13             Get process P ∈ P incident with channel E+;
14             P_inst ← P ;
15             D_{P_inst} ← R+_E(D_{snk_inst}) ;
16             P' ← P' ∪ P_inst ;
17         P' ← P' ∪ P_snk_inst ;
18 foreach P_inst ∈ P' do
19     E_inst ← instantiateChannels(P_inst, E) ;
20     E' ← E' ∪ E_inst;
```

domain $D_{P_{snk\_inst}}$. If this iteration $\vec{K}$ transitively depends on other iterations in domain D_{snk} (line 9), then domain $D_{P_{snk\_inst}}$ contains also all iterations in D_{snk} that iteration $\vec{K}$ depends on. All such iterations can be computed by *slicing* a transitive dependence R^+ using iteration $\vec{K}$, denoted as $R^+(\vec{K})$. It is formally defined as:

$$R^+(\vec{K}) = \{\vec{J} \mid \vec{I} \rightarrow \vec{J} \in R^+ \wedge \vec{I} = \vec{K}\},$$

where $\vec{K}$ is a constant vector (see an example in Equation (9)). Therefore, in this case, we can obtain $D_{P_{snk\_inst}}$ as shown at line 10 in Algorithm 1. In contrast, if the iteration $\vec{K}$ does not depend on any other iteration in D_{snk}, then $D_{P_{snk\_inst}}$ is simply equal to $\vec{K}$ (line 12). Also, in this case, $\vec{K}$ transitively depends on another PPN process P through transitive dependence relation R^+_E, where $P \neq P_{snk}$. Therefore, we need to instantiate a process instance P_{inst} for process P (lines 13–15). Domain $D_{P_{inst}}$ can be computed by *applying* domain $D_{P_{snk\_inst}}$ to dependence relation R^+_E, denoted as $R^+_E(D_{P_{snk\_inst}})$. $R^+_E(D_{P_{snk\_inst}})$ is defined as:

$$R^+_E(D_{P_{snk\_inst}}) = \left\{\vec{J} \mid \vec{I} \rightarrow \vec{J} \in R^+_E \wedge \vec{I} \in D_{P_{snk\_inst}}\right\}.$$

An example of the apply operation can be seen in Equation (11). Finally, all process instances in the same communication-free partitions can be instantiated (lines 5–17).

Once all process instances for each communication-free partition are instantiated, as the next step, we need to instantiate channels for all process instances (line 19 in Algorithm 1). The procedure of instantiating all channels for a process instance is depicted in Algorithm 2. As input, it takes a process instance P_{inst} with constructed domain $D_{P_{inst}}$ and all channels $\mathcal{E}$ in the initial PPN. The algorithm outputs a set of channels $\mathcal{E}_{inst}$ incident with process instance P_{inst}. In Algorithm 2, if both the input

ALGORITHM 2: Procedure *instantiateChannels*

Input: A process instance P_{inst} and a set channels $\mathcal{E}$.
Result: A set of channels $\mathcal{E}_{inst}$ incident with process instance P_{inst}.

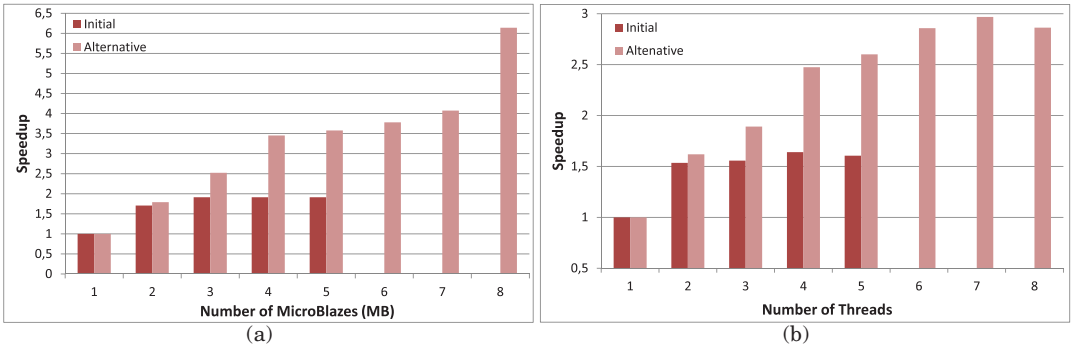
1 Get $D_{P_{inst}}$ of P_{inst};
2 **foreach** *Channel* $E \in \mathcal{E}$ *incident with* P_{inst} **do**
3 Get D_{IP} and D_{OP} associated with channel E;
4 $E_{inst} \leftarrow E$;
5 **if** $D_{IP} \cap D_{P_{inst}} \neq \emptyset$ **and** $D_{OP} \cap D_{P_{inst}} \neq \emptyset$ **then** /* a self-channel */
6 $D_{IP\_inst} \leftarrow D_{inst} \cap D_{IP}$, $\mathrm{dom}R_{E_{inst}} \leftarrow D_{IP\_inst}$;
7 $D_{OP\_inst} \leftarrow D_{inst} \cap D_{OP}$, $\mathrm{ran}R_{E_{inst}} \leftarrow D_{OP\_inst}$;
8 $\mathcal{E}_{inst} \leftarrow \mathcal{E}_{inst} \cup E_{inst}$;
9 **else if** $D_{IP} \cap D_{P_{inst}} \neq \emptyset$ **and** $D_{OP} \cap D_{P_{inst}} = \emptyset$ **then** /* an incoming channel */
10 $D_{IP\_inst} \leftarrow D_{inst} \cap D_{IP}$, $\mathrm{dom}R_{E_{inst}} \leftarrow D_{IP\_inst}$;
11 $\mathcal{E}_{inst} \leftarrow \mathcal{E}_{inst} \cup E_{inst}$;
12 **else if** $D_{IP} \cap D_{P_{inst}} = \emptyset$ **and** $D_{OP} \cap D_{P_{inst}} \neq \emptyset$ **then** /* an outgoing channel */
13 $D_{OP\_inst} \leftarrow D_{inst} \cap D_{OP}$, $\mathrm{ran}R_{E_{inst}} \leftarrow D_{OP\_inst}$;

port and output port of a channel E are incident with P_{inst} (line 5), a new self-channel E_{inst} is instantiated with the corresponding input and output port domains (lines 6 and 7). An example of instantiating self-channel $E33_{inst}$ for process instance $P3_{inst}$ can be seen in Equation (12). If only the input port or output port of a channel E is incident with P_{inst} (line 9 and 12 respectively), it denotes a dependence relation from/to another process instance in the same communication-free partition. In other words, it is either an incoming or outgoing channel of process instance P_{inst}. In this case, we instantiate only one channel with its corresponding input and output port domains (lines 10 and 13). Therefore, using Algorithm 2, we can instantiate all channels incident with a process instance P_{inst}.

3. EXPERIMENTAL RESULTS

In this section, we present the performance results obtained by applying our approach explained in Section 2 and prototyping two real-life streaming applications on two different platforms. Then, we present a set of experiments to evaluate the time complexity of our approach.

We selected two different platforms, a Xilinx ML605 board equipped with a Virtex 6 FPGA (referred as FPGA platform hereinafter) and a desktop multi-core platform containing an Intel i7-920 processor running at 2.66GHz with 4 cores and 4GB system memory (referred as desktop platform hereinafter). For the FPGA platform, the generated MPSoCs consist of up to 8 MicroBlaze (MB) soft-cores interconnected via Xilinx' Fast Simplex Link FIFOs. All MBs run at 100Mhz with their own 64KB program memory and 64KB data memory. On the desktop platform, a main thread was used to measure the performance and to spawn up to 8 threads, due to hyper-threading. The inter-core data communication cost on the desktop platform is much higher than that on the FPGA platform. Therefore, the performance gain introduced using our approach was evaluated on the platforms with different computation/communication characteristics. We conducted all experiments using the open-source ESPAM [Nikolov et al. 2008] tool, the Xilinx Platform Studio 13.2, and Microsoft Visual Studio 2008. All generated programs were compiled using compilers `mb-g++4.1.2` and `g++4.52` on the selected platforms respectively, with optimization level `-O2`.



(a) (b)

Fig. 7. Performance results of mapping the MJPEG encoder onto (a) FPGA-based MPSoC platforms and onto (b) a desktop multi-core platform.

3.1. Case Studies

We considered two real-life applications modeled using the PPN MoC, namely a Motion-JPEG (MJPEG) encoder used in Cong et al. [2009] and the FM radio application taken from the StreamIT benchmark suite [Gordon et al. 2006]. The MJPEG encoder encodes frames with size 128×128 pixels. For the FM radio application, we took the provided sequential C implementation to generate the initial PPN with the following parameters: decimation rate 4, tap size 64, and 10 equalization bands. To optimally balance the workloads across a particular number of PEs, we exhaustively mapped all possible groupings of the obtained communication-free partitions on both platforms. As a reference, we also implemented the initial PPNs of both applications on the selected platforms by performing maximal load-balancing and optimal pipelining, such that the best possible mapping was found for a given number of MBs or threads. The metric used to evaluate the performance results is the relative speedup compared to the 1-MB or 1-thread system implementation.

The performance results of mapping the MJPEG encoder are plotted in Figure 7(a) for the FPGA platform and in Figure 7(b) for the desktop platform. As expected, the implementation on the desktop platform results in less speedup than the one obtained on the FPGA platform for the same number of MBs or threads in use. This is because of the shared memory architecture and very costly inter-thread communication on the desktop platform. Also, the initial PPN mapped onto the desktop platform using 1 thread is already highly optimized by the compiler. For the mapping of the initial PPN (denoted as *Initial*), the initial PPN does not have enough processes to utilize more than 5 MBs or threads. It can be seen that up to $1.91X$ speedup for the FPGA platforms and $1.64X$ speedup for the desktop platform are achieved. The main reason is that the workloads of processes in the initial PPN are not well-balanced, as the Discrete Cosine Transform (DCT) dominates the total execution time of the MJPEG encoder. Although all PPN processes are fully pipelined, the speedup is limited by the longest pipeline stage, the DCT process. For the desktop platform, the pipelining leads to less benefits compared to the FPGA platform, because the communication between threads mapped onto different cores cannot be completely overlapped by computation.

Compared to the mapping of the initial PPN for the MJPEG encoder, our approach (denoted as *Alternative* in Figure 7(a) and 1(b)) leads to better performance. Our approach outperforms the mapping of the initial PPN by 5% to 87.05% on 2 to 5 MBs. As shown in Figure 7(a) for the FPGA platform, the speedup increases linearly for the mapping of the alternative PPN onto 1 to 4 MBs ($3.45X$ speedup on 4 MBs). In case of 5 to 7 MBs, the speedup increases only slightly ($3.6X$ to $4.09X$ speedup on

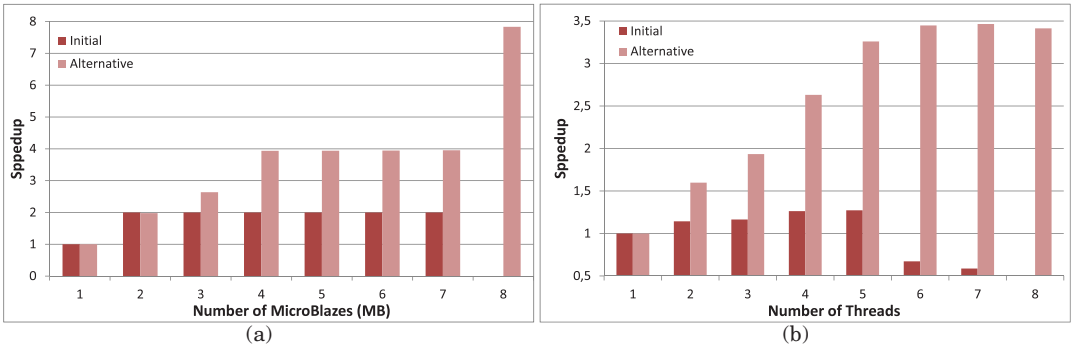


Fig. 8. Performance results of mapping the FM radio application onto (a) FPGA-based MPSoC platforms and onto (b) a desktop multi-core platform.

5 to 7 MBs). We found that unbalanced workloads and the single data sink become bottlenecks for these cases. As the number of MBs increases, a slightly unbalanced grouping of communication-free partitions has large impact on the performance. As a consequence, the single data sink is constantly blocking on the group of partitions with the heaviest workload. Of course, modern architectures may have multiple I/O ports, namely multiple data sinks. For instance, the authors in [Gordon et al. 2006] observe 18.4% performance improvement on the 16-core RAW architecture with 16 data sinks compared to the one with the single data sink. In the best case, our approach results in 6.14*X* speedup on 8 MBs, when the grouping of the obtained partitions balances the workload across 8 MBs. For the results on the desktop platform shown in Figure 7(b), the mapping of the alternative PPN outperforms the mapping of the initial PPN by 5.5% to 61.97% using 2 to 5 threads. Moreover, the effect of unbalanced grouping of communication-free partitions is amortized by the higher communication cost compared to the FPGA platform. In the best case, 2.97*X* speedup is achieved using 7 threads. When 8 threads are used, the main thread, mentioned earlier, introduces extra overhead. Therefore, the 8-thread implementation performs 3.68% worse than the 7-thread implementation.

For the FM radio application, the workloads of PPN processes in the initial PPN are overall not balanced. The low pass and high pass filters in the equalizer dominate the total execution time of the application. Moreover, the communication between PPN processes is performed at more fine-grained level compared to the MJPEG encoder, i.e., at each iteration, one audio sample is flowed through all PPN processes instead of one macroblock as in the MJPEG encoder. The obtained speedup of mapping the initial PPN (denoted as *Initial*) is plotted in Figure 8(a) for the FPGA platform and in Figure 8(b) for the desktop platform. In the best case on the FPGA platform, by pipelining all processes in the initial PPN and offloading the high pass filter (or low pass filter) in the equalizer to a separate MB, 1.99*X* speedup is achieved on 2 MBs. On the desktop platform shown in Figure 8(b), the best mapping of the initial PPN is found using 5 threads occupying 4 cores, i.e., 1.27*X* speedup. In case of 6 and 7 threads, the implementation slows down compared to the 1-thread implementation. The fine-grained communication and the little workloads of some threads (e.g., the Demodulation and the Amplify processes in the Equalizer) fully expose the communication/synchronization overhead which dominates the total execution time.

After communication-free partitioning, the alternative PPN of the FM radio application exhibits ample data-level parallelism. Also, the fine-grained communication between MBs or threads in the initial PPN is completely eliminated, except the

Table II. Execution Time on Benchmarks

Benchmark	$\|\mathcal{P}\|$	$\|\mathcal{E}\|$	Array dimensions	*Execution time (sec.)*
`adi`	12	67	3	2.644
`gramschmidt`	8	19	2	0.924
`fdtd-2d`	9	27	2	0.604
`correlation`	12	20	2	0.076
`reg-detect`	8	11	3	0.068
`dynprog`	8	12	3	0.064
`gauss-filter`	11	18	2	0.044
`covariance`	8	11	2	0.032

communication from the data source and to the data sink. For the results on the FPGA platform shown in Figure 8(a) (denoted as *Alternative*), the obtained speedup by mapping the alternative PPN outperforms mapping the initial PPN by 32.05% to 97.74% on 3 to 7 MBs. Compared to the 4-MB implementation, the mapping of the alternative PPN onto 5 to 7 MBs does not result in further improvements. This is because, as the number of MBs increases, the workloads of the obtained communication-free partitions cannot be evenly distributed. This fact combined with the relatively cheaper inter-MB communication on the FPGA platform, shows that our communication-free partitioning does not bring too much benefits on 5 to 7 MBs. Once the workload is balanced, 7.83X speedup is achieved on 8 MBs. On the desktop platform, our approach (denoted as *Alternative* in Figure 8(b)) outperforms the mapping of the initial PPN by 39.79% to 489.27% using 2 to 7 threads. In the best case, speedup 3.46X is observed using 7 threads. The 8-thread implementation performs 1.51% worse compared to the 7-thread one due to the overhead introduced by the main thread similar to the MJPEG case study.

3.2. Time Complexity of Our Approach

To quantify the time complexity of our approach, we conducted experiments on a set of real-life benchmarks from Polybench [Polybench 2012]. Other benchmarks are less complex than the benchmarks listed in Table II in terms of their characteristics. The characteristics of each benchmark are given in columns 2 to 4 in Table II. The benchmarks differ in the of number of PPN processes (denoted by $|\mathcal{P}|$) and channels (denoted by $|\mathcal{E}|$) in the initial PPNs, as well as dimensions of data arrays accessed in PPN processes. For instance, the Alternating Direction Implicit (`adi`) solver in Table II operates on 3 dimensional data arrays. In practice, it can be seen that, from the last column in Table II, our approach takes less than 3 seconds to derive all communication-free partitions for the considered benchmarks. This shows that our approach is very fast even for relatively large PPNs such as the PPN of the `adi` application.

4. CONCLUSIONS

In this article, we have shown that the mapping of streaming applications considering a single initial application specification cannot fully utilize the processing power of MPSoC platforms. Using the Polyhedral Process Network (PPN) MoC as the application specification, we have presented an analytical framework to determine the maximum data-level parallelism, i.e., the maximum number of communication-free partitions. Subsequently, we have proposed an approach to transform an initial PPN to a set of communication-free partitions, if it exists. The experimental results on FPGA-based MPSoCs and desktop multi-core platforms showed that our approach leads to significantly better performance than the approaches, in which alternative application specifications are not taken into account.

We are also aware of some spaces to improve the applicability of our approach. First, a better exploration strategy is required to automatically select alternative application specifications with respect to the target platform, i.e., a given number of PEs available. Second, we found out that a holistic mapping of streaming applications also strongly depends on the platform characteristics. In case of relatively low communication cost between PEs, by allowing a certain degree of communication between PEs, a more load-balanced mapping could lead to better system performance. Third, the performance improvement resulted from using our approach comes at the cost of increased memory usage. Hence, we would like to investigate the trade-off between memory usage and obtained partitions in the future. And, if one partition does not entirely fit to one PE in terms of the memory requirement, we would like to develop a re-partitioning strategy while introducing least amount of communication between PEs possible.

REFERENCES

Cong, J., Gururaj, K., Han, G., and Jiang, W. 2009. Synthesis algorithm for application-specific homogeneous processor networks. *IEEE Trans. VLSI. Syst. 17*, 1318–1329.

Feautrier, P. 1991. Dataflow analysis of array and scalar references. Int. J. Parallel Program 20, 1, 23–53.

Feautrier, P. 1996. Automatic parallelization in the polytope model. In *The Data Parallel Programming Model*, Springer-Verlag, 79–103.

Gerstlauer, A., Haubelt, C., Pimentel, A., Stefanov, T., Gajski, D., and Teich, J. 2009. Electronic system-level synthesis methodologies. *IEEE. Trans. Comput. Aid. Des. Integrat. Circuits. Syst. 28,* 10, 1517–1530.

Gerstlauer, A., Peng, J., Shin, D., Gajski, D., Nakamura, A., Araki, D., and Nishihara, Y. 2008. Specify-explore-refine (SER): from specification to implementation. In *Proceedings of the 45th Annual Design Automation Conference*. ACM, New York, NY, 586–591.

Gordon, M. I., Thies, W., and Amarasinghe, S. 2006. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of The 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 151–162.

Grosser, T., Zheng, H., A, R., Simbürger, A., Grösslinger, A., and Pouchet, L.-N. 2011. Polly - polyhedral optimization in LLVM. In *Proceedings of the 1st International Workshop on Polyhedral Compilation Techniques*.

Kahn, G. 1974. The semantics of a simple language for parallel programming. In *Proceedings of the Information Processing*. North-Holland.

Kelly, W., Pugh, W., Rosser, E., and Shpeisman, T. 1996. Transitive closure of infinite graphs and its applications. *Int. J. Parallel Program. 24*, 579–598.

Kudlur, M. and Mahlke, S. 2008. Orchestrating the execution of stream programs on multicore platforms. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, NY, 114–124.

Lee, E. A. and Messerschmitt, D. G. 1987. Synchronous data flow. *Proc. IEEE 75,* 9, 1235–1245.

Liao, S.-w., Du, Z., Wu, G., and Lueh, G.-Y. 2006. Data and computation transformations for brook streaming applications on multiprocessors. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, Los Alamitos, CA, 196–207.

Meijer, S., Nikolov, H., and Stefanov, T. 2010. Combining process splitting and merging transformations for polyhedral process networks. In *Proceedings of the 8th* International *IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia'10)*. 97–106.

Nikolov, H., Stefanov, T., and Deprettere, E. 2008. Systematic and automated multiprocessor system design, programming, and implementation. *IEEE Trans. Comput. Aid. Des. Integrat. Circuits Syst. 27*, 542–555.

Pimentel, A. D., Erbas, C., and Polstra, S. 2006. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Trans. Comput. 55*, 99–112.

Polybench. 2012. The Polyhedral Benchmark Suite. http://www.cse.ohio-state.edu/pouchet/software/polybench/.

Pugh, W. and Rosser, E. 1997. Iteration space slicing and its application to communication optimization. In *Proceedings of the 11th International Conference on Supercomputing*. ACM, New York, NY, 221–228.

Stuijk, S., Basten, T., Geilen, M. C. W., and Corporaal, H. 2007. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *Proceedings of the 44th Annual Design Automation Conference*. ACM, New York, NY, 777–782.

THIELE, L., BACIVAROV, I., HAID, W., AND HUANG, K. 2007. Mapping applications to tiled multiprocessor embedded systems. In *Proceedings of the 7th* International *Conference on Application of Concurrency to System Design (ACSD)*. IEEE Computer Society, 29–40.

VERDOOLAEGE, S. 2010. isl: An integer set library for the polyhedral model. Lecture Notes in Computer Science, K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, Eds., Springer, 299–302.

VERDOOLAEGE, S., CATTHOOR, F., BRUYNOOGHE, M., AND JANSSENS, G. 2003. Multi-dimensional incremental loop fusion for data locality. In *Proceedings of the Conference on Application-Specific Systems, Architectures and Processors*. IEEE, 17–27.

VERDOOLAEGE, S., NIKOLOV, H., AND STEFANOV, T. 2007a. pn: a tool for improved derivation of process networks. *EURASIP J. Embed. Syst. 2007*, 13.

VERDOOLAEGE, S., SEGHIR, R., BEYLS, K., LOECHNER, V., AND BRUYNOOGHE, M. 2007b. Counting integer points in parametric polytopes using barvinok's rational functions. *Algorithmica 48*, 37–66.

YANG, H. AND HA, S. 2009. Pipelined data parallel task mapping/scheduling technique for MPSoC. In *Proceedings of 12th International Conference on Design, Automation and Test in Europe*. 69–74.

ZHU, J., SANDER, I., AND JANTSCH, A. 2010. Constrained global scheduling of streaming applications on MPSoCs. In *Proceedings of the Asia and South Pacific Design Automation Conference*. IEEE Press, 223–228.

Received June 2012; revised August 2012; accepted September 2012