Adaptive streaming applications : analysis and implementation models

Zhai, J.T.

Chapter 4

Exploiting Maximum Data-level Parallelism without Inter-processor Communication

Jiali Teddy Zhai, Hristo Nikolov, and Todor Stefanov, "Mapping of Streaming Applications considering Alternative Application Specifications", in ACM Transactions on Embedded Computing Systems (TECS), vol. 12, Issue 1s, Article 34, March 2013.
Jiali Teddy Zhai, Hristo Nikolov, Todor Stefanov, "Mapping Streaming Applications considering Alternative Application Specifications (Extended Abstract)", *In Proceedings of the 10th IEEE Symposium on Embedded System for Real-Time Multimedia (ESTIMedia'12)*, Tampere, Finland, October 11-12, 2012.

As explained in Section 1.1.4, during the synthesis step of a model-based design methodology, all possible combinations of Processing Element (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, existing DSE approaches search the design space using different algorithms, e.g., stepwise refinement in [51], heuristics in [109] and [111], evolutionary algorithms in [100, 115], branch-and-bound in [34], and constraint programming in [139]. These DSE approaches consider only a single application specification given by application designers.

As mentioned in Section 1.2, an application specification given by application designers often does not take into account the underlying computation and communication capability of an MPSoC. Indeed, the authors in [71] 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.

The discussion above indicates that alternative application specifications may be needed for efficient mapping of an application. In this thesis, 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. Ideally, 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, mapping these n tasks onto n PEs will always result in n times speedup. In this case, all PEs are equally loaded and 100% utilized without the need to synchronize and communicate data with each other.

In this chapter, we study the problem of whether an alternative PPN exists for an initial PPN, which consists of only independent and load-balanced processes. Specifically, we divide the problem into two stages. In the first stage, we analytically identify independent execution of PPN processes, 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, given n PEs, the application mapping problem is considered as grouping the set of obtained communication-free partitions to balance the application workloads across all PEs, such that the resulting performance (i.e., throughput) is maximized. 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.

Scope of Work

In this chapter, we consider streaming applications which can be modeled using the PPN MoC. We assume that there are only one source and sink processes respectively and they are orders of magnitude faster than the remaining processes that perform computation. The source and sink processes represent environment and are thus not partitioned. Furthermore, the achievable performance of a PPN is not constrained by the buffer size required for each communication edge. It is possible to compute a buffer size for each PPN edge using the PNgen compiler such that larger buffer sizes do not increase the performance. We statically allocate a FIFO buffer for each PPN edge on target platforms. The target platforms considered in this chapter 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.

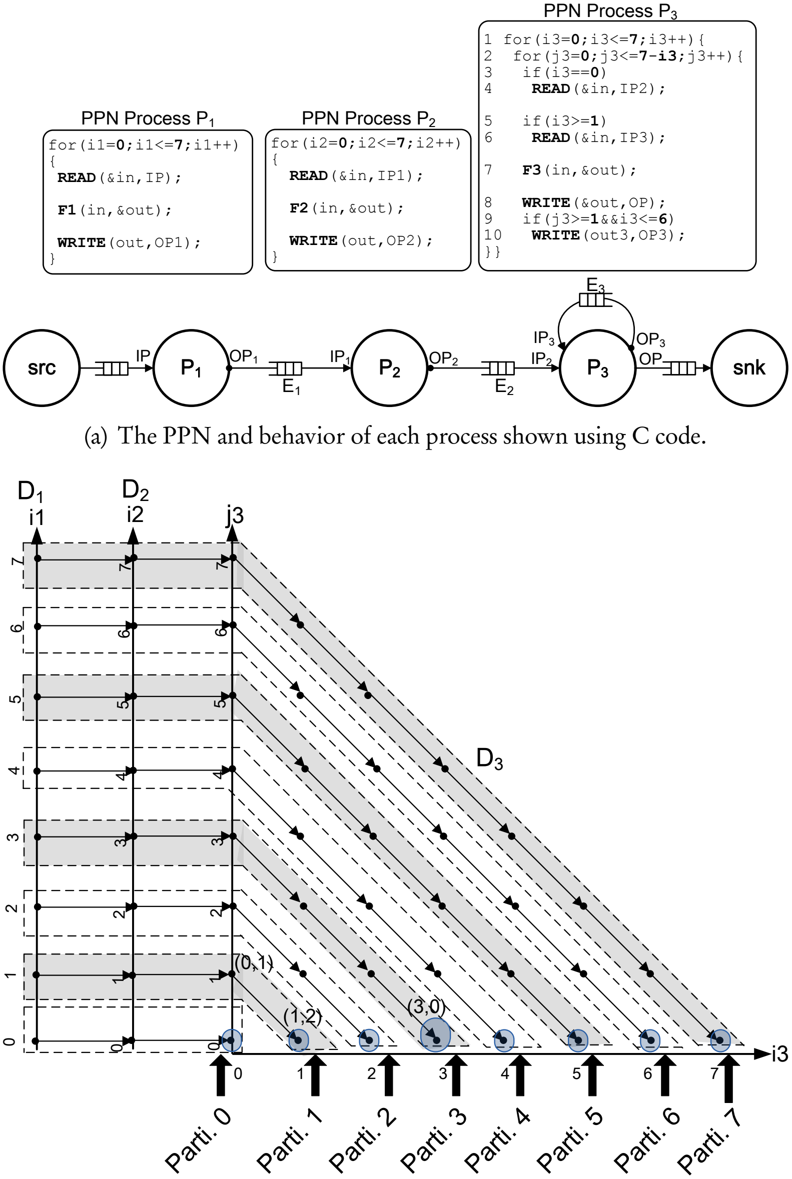
4.1 Motivating Example

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 4.1(a). This example is used throughout this chapter as a running example. The PPN represents a common topology of a parallel application specification and consists of three PPN processes P_1, P_2, and P_3 communicating data via FIFO edges. Note that P_3 has cyclic data dependences through edge E_3. The behavior of each PPN process is given as C code above the corresponding process. Besides the PPN processes expressing the application behavior, the processes src and snk represent the environment which provides input data and collects results. Their execution is expressed by two domains, namely $D_{src} = D_{IP}$ and $D_{snk} = D_{OP} = D_3$. For instance, D_{snk} is given as

$$D_{snk} = \{(i3, j3) \in \mathbb{Z}^2 \mid 0 \leq i3 \leq 7 \land 0 \leq j3 \leq 7 - i3\}. \tag{4.1}$$

Suppose that processes src and snk are much faster than the PPN processes and the PPN is to be mapped onto the platform shown in Figure 4.2. The workloads of functions F1, F2, and F3 in Figure 4.1(a) on the PEs are 6, 100, and 30 time units, respectively. Communication latency via the interconnection structure is assumed to be 5 time units and communication latency through local memory is considered as negligible. 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 4.1(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 P_1 and P_2 are pipelined and mapped onto PE1, while process P_3 is mapped onto PE2 as shown in Figure 4.2. Figure 4.3 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 4.1(a) is visualized in Figure 4.1(b). The dots represent individual iterations of the PPN processes. For example, one iteration of P_3 comprises one execution of its loop body (lines 3 - 10 of P_3 in Figure 4.1(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 4.1(b)



PPN Process P1

```
for(i1=0;i1<=7;i1++)
{
 READ(&in,IP);

 F1(in,&out);

 WRITE(out,OP1);
}
```

PPN Process P2

```
for(i2=0;i2<=7;i2++)
{
 READ(&in,IP1);

 F2(in,&out);

 WRITE(out,OP2);
}
```

PPN Process P3

```
for(i3=0;i3<=7;i3++){
 for(j3=0;j3<=7-i3;j3++){
  if(i3==0)
   READ(&in,IP2);

  if(i3>=1)
   READ(&in,IP3);

  F3(in,&out);

  WRITE(&out,OP);
  if(j3>=1&&i3<=6)
   WRITE(out3,OP3);
}}
```

(a) The PPN and behavior of each process shown using C code.

(b) Process domains of all PPN processes and 8 communication-free partitions.

Figure 4.1: An example of a PPN and its communication-free partitions.

(each partition is surrounded by a dashed box). One can see in Figure 4.1(b) that no arrows (data dependences) exist across the partitions. Each partition contains a subset

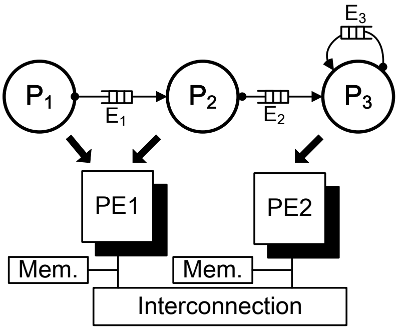


Figure 4.2: Mapping of the PPN in Figure 4.1(a) onto 2 PEs achieving the maximum performance.

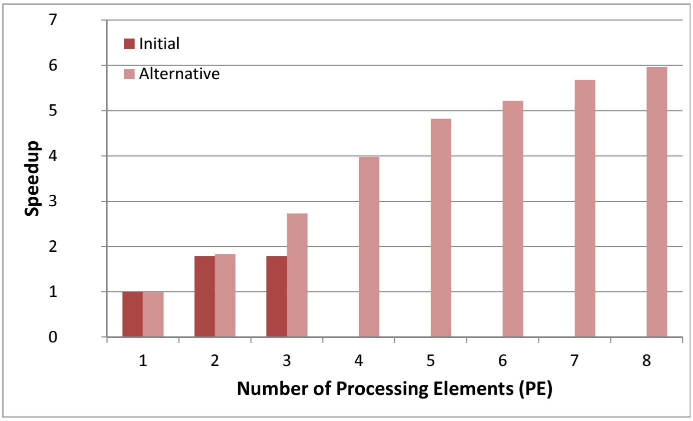
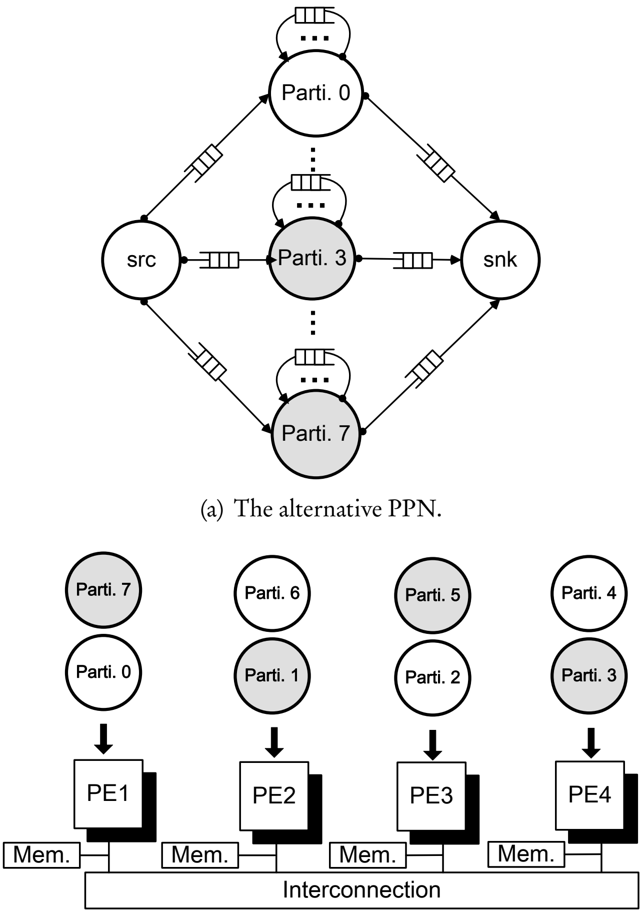


Figure 4.3: Performance results of mapping the initial PPN and the alternative PPN after communication-free partitioning.

execution of PPN processes P_1, P_2, and P_3 in Figure 4.1(a). After communication-free partitioning, the initial PPN in Figure 4.1(a) is transformed to the alternative PPN shown in Figure 4.4(a). The only communication between PEs occurs when input data is demultiplexed from process src to all partitions and output produced by the partitions is multiplexed to process snk. For example, this can be seen with the help of Figure 4.1(b). In the initial PPN, process src sends the input data to P_1 at its iterations from (0) to (7) due to a dependence relation (see Definition 2.1.7 on page 26). In the alternative PPN, with the same dependence relation, process src sends the input data at iteration (0) of P_1 to partition *Parti. 0*, the input data at iteration (1) of P_1 to partition *Parti. 1*, and so on. Analogously, in the alternative PPN, process snk collects the output data produced at iteration (0, 0) of P_3 from



(a) The alternative PPN.

(b) Mapping of the alternative PPN onto 4 PEs (the data source and sink as well as all edges connected to both of them are omitted for succinctness).

Figure 4.4: The PPN in Figure 4.1(a) after communication-free partitioning and its mapping.

partition *Parti. 0*, the output data produced at iterations $(0,1)$ and $(1,2)$ of P_3 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. Except the communication 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 4.4(a) is shown in Figure 4.4(b).

Figure 4.3 also shows the achieved speedup of pareto-optimal mappings of the alternative PPN in Figure 4.4(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 4.3 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, 4 groups with 2 partitions each shown in Figure 4.4(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.

4.2 Related Work

An alternative application specification modeled as a SDF graph is considered in [133]. To exploit better parallelism in the SDF graph, all actors in the initial SDF graph are converted to their equivalent Homogeneous SDF actors (all production/consumption rates equal to 1). The conversion may lead 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 [133], we consider a more expressive MoC than SDF, i.e., the PPN MoC. Also, instead of completely unfolding all PPN processes (equal to unfolding actors in [133]), we operate on a compact representation which avoids the explosion in the size of the graph. Moreover, this compact representation also allows us to analytically determine the maximum amount of DLP, i.e., the maximal number of communication-free partitions.

Similar to [133], SDF is also used as the underlying MoC in [54]. Each SDF actor is furthermore restricted 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. Conceptually, this method also aims at extracting DLP without communication between the compound actors. Compared to [54], the PPN MoC considered in this chapter is more general with an arbitrary number of input and output ports of PPN processes. The problem addressed in this chapter is thus more difficult as simple fusion-duplication is not applicable to PPN processes. Also, *stateful* actors (see for instance process P_3 in Figure 4.1(a)) cannot be fused and duplicated in [54]. Instead, software pipelining techniques are applied to the stateful actors. Software pipelining brings performance improvement assuming that communication latency between different PEs, on which different pipeline stages are assigned, could be completely overlapped by computation. However, we believe that the communication latency may not be hidden and completely overlapped by computation, especially considering emerging MPSoC platforms interconnected via NoCs as motivated in Section 1.1.3. In contrast, our approach tries to extract maximum DLP even for the PPN processes with cyclic data dependences while completely avoiding communication between PEs. This parallelization strategy may fit better future MPSoC platforms with increasingly larger communication latency.

The PPN MoC is used in [86]. 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 chapter, we propose a systematic procedure to partition and merge PPN processes in a PPN.

In [78], 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.

4.3 Finding all Dependences in a PPN

For streaming applications, input data is read from the data source (i.e., environment), subsequently processed by PPN processes at their iterations during the execution, and finally written to the data sink. Recall that a PPN produces output to the environment, represented as a sink process whose domain is denoted as D_{snk}. The output produced at an iteration $\vec{I} \in D_{snk}$ directly or indirectly depends on several iterations of PPN processes. If two dependent iterations mapped onto different PEs, inter-PE communication will take place. 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 as defined in Definition 2.1.7 on page 26. For example, Figure 4.5 illustrates the process domain D_3 of process P_3 in Figure 4.1(a). Dependence relation $R_3 = \{(1,2) \rightarrow (0,3)\}$ in Figure 4.5 (the bold arrow) expresses a direct dependence. In contrast, iteration

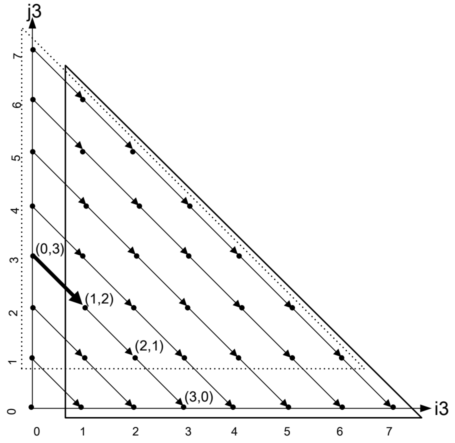


Figure 4.5: Domain of PPN process P_3 in Figure 4.1(a). The input port domain of IP_3 (surrounded by the solid triangle), output port domain of OP_3 (surrounded by the dotted triangle), and dependence relation R_3 (denoted by the arrows between dots).

(2,1) indirectly depends on iteration (0,3) through iteration (1,2). In this chapter, we formulate the problem of finding all direct and indirect data dependences by computing *transitive closure* [65, 102], denoted by R^+, of affine dependence relation R. It is formally defined as:

$$\vec{I} \rightarrow \vec{J} \in R^+ \Leftrightarrow (\vec{I} \rightarrow \vec{J}) \in R \vee \exists \vec{K} \text{ s.t. } (\vec{I} \rightarrow \vec{K}) \in R \wedge (\vec{K} \rightarrow \vec{J}) \in R^+. \tag{4.2}$$

From Equation (4.2), 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 [65]. In contrast, we consider an affine over-approximation in case of non-affine closed form. First, the over-approximation guarantees that a valid schedule always can be found for each communication-free partition, but at the cost of potentially fewer communication-free partitions. Second, existing powerful code generation methods [25] 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:

$$R_{deps} = \bigcup_{\forall E_i \in \mathcal{E}} R_i,$$

where $\mathcal{E}$ is the set of edges in the PPN. Subsequently, we can compute the transitive closure of the union R_{deps}. In this chapter, we use the *isl* [123] library to compute the transitive closure of affine dependence relations in a potentially over-approximated closed form. For the PPN in Figure 4.1(a), computing the union of all dependence relations yields:

$$R_{deps} = R_1 \cup R_2 \cup R_3.$$

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

$$R^+_{deps} = R_{deps} \cup R^+_{13} \cup R^+_{23} \cup R^+_{33},$$

where R^+_{13}, R^+_{23}, and R^+_{33} are transitive dependence relations, represented as follows:

$$R^+_{13} = \{(i3, j3) \rightarrow (i1) \mid 0 \leq i3 \leq i1 \wedge i1 \leq 7 \wedge i1 = i3 + j3\}, \tag{4.3a}$$

$$R^+_{23} = \{(i3, j3) \rightarrow (i2) \mid 0 \leq i3 \leq i2 \wedge i2 \leq 7 \wedge i2 = i3 + j3\}, \tag{4.3b}$$

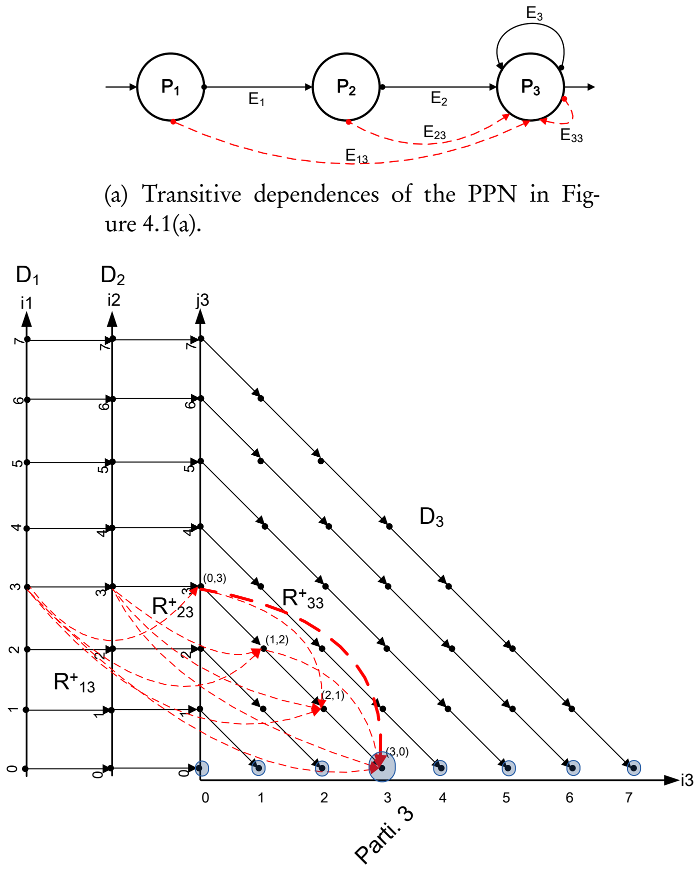
$$\begin{aligned} R^+_{33} = \{(i3, j3) \rightarrow (i3', j3') \mid & 1 \leq i3 \leq 7 \wedge 0 \leq j3 \leq 7 - i3 \wedge 0 \leq i3' \leq 6 \\ & \wedge 0 \leq i3' \leq i3 + j3 - 1 \wedge j3' = i3 + j3 - i3'\}. \end{aligned} \tag{4.3c}$$

After computing the transitive closure of all dependence relations in the PPN in Figure 4.1(a), 3 extra edges E_{13}, E_{23}, and E_{33} corresponding to the transitive dependence relations are added in the PPN as shown in Figure 4.6(a). For the execution of the PPN (domains of PPN processes P_1, P_2, and P_3) shown in Figure 4.1(b), a set of transitive dependences is illustrated as dashed arrows in Figure 4.6(b). For instance, $R^+_{33} = \{(3,0) \rightarrow (0,3)\}$, shown as the bold and dashed arrow, indicates that iteration $(3,0)$ of PPN process P_3 transitively depends on iteration $(0,3)$ of itself.

4.4 Computing the Number of Communication-free Partitions

As explained in Section 4.3, 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.

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 4.1(b)). Consider the PPN in Figure 4.1(a) and its execution illustrated in Figure 4.1(b). As explained in Section 4.1, $D_{snk} = D_3$ (see the triangular part in Figure 4.1(b) denoted as D_3). Our goal is to find the 8 iterations marked by circles in Figure 4.1(b). It can be seen that they are independent



(a) Transitive dependences of the PPN in Figure 4.1(a).

(b) The set of transitive dependences for communication-free partition *Parti. 3* in Figure 4.1(b).

Figure 4.6: Finding transitive dependences of the PPN.

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 4.4.1. *For a PPN, any transitive dependence relation*

$$R_i^+ = \{\vec{I} \rightarrow \vec{J} \mid \vec{I} \in D_{IP} \wedge \vec{J} \in D_{OP}\}$$

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[1]. 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 (4.2). If there exists $\vec{I} \to \vec{K} \in R$ and $\vec{K} \to \vec{J} \in R$, both iterations $\vec{I}$ and $\vec{K}$ are mapped to iteration $\vec{J}$ in R^+. ∎

For instance, transitive relation R_{23}^+ in Figure 4.6(b) denotes that iterations $(0,3)$, $(1,2)$, $(2,1)$, and $(3,0)$ of P_3 are mapped to iteration (3) of P_2. That is, R_{23}^+ maps the iterations $\vec{I} \in D_{IP_2} \cup D_{IP_3}$ to the iterations $\vec{J} \in D_{OP_2}$ shown in Figure 4.1(a).

In addition to Lemma 4.4.1, we introduce a definition, called *independent sink domain*, denoted by D_{snk}^{ind}.

Definition 4.4.1 (Independent Sink Domain)**.** The independent sink domain D_{snk}^{ind} for a PPN is a subset of the process domain of the sink process D_{snk}, namely $D_{snk}^{ind} \subseteq D_{snk}$. The following condition holds for any two iterations $(\vec{I},\vec{J}) \in D_{snk}^{ind}$, where $\vec{I} \neq \vec{J}$:

$$\neg\exists(\vec{I},\vec{J}) \in D_{snk}^{ind} \; : \; \vec{I} \to \vec{J} \in R^+. \tag{4.4}$$

D_{snk}^{ind} is given by

$$\begin{aligned} D_{\text{snk}}^{\text{ind}} = & \\ & \{\vec{I} \in \mathbb{Z}^d \mid \exists R^+ : \vec{I} \to \vec{J} \in R^+ \wedge \vec{I} \in D_{snk} \wedge \vec{J} \in D_{snk} \wedge \vec{I} \in (\text{dom}R^+ - \text{ran}R^+)\} \qquad (4.5a) \\ & \bigcup \\ & \{\vec{I} \in \mathbb{Z}^d \mid \forall R^+ : \vec{I} \to \vec{J} \in R^+ \wedge \vec{I} \in D_{snk} \wedge \vec{J} \notin D_{snk} \wedge \vec{I} \in \text{dom}R^+\}, \qquad (4.5b) \end{aligned}$$

where domR^+ is the domain of transitive relation R^+ and ranR^+ is the range of transitive relation R^+.

The condition in Equation (4.4) states that the iterations in D_{snk}^{ind} are not transitively dependent on each other.

Based on Lemma 4.4.1 and Definition 4.4.1, we can have the following theorem.

[1] The PPNs with these two properties are called *live*.

Theorem 4.4.1. *For any PPN, the number of communication-free partitions is equal to* $|D_{snk}^{ind}|$.

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 $\vec{I} \rightarrow \vec{J} \in R^+ \wedge \vec{J} \in D_{snk}$ in Equation (4.5a), whereas the latter condition is expressed as $\vec{I} \rightarrow \vec{J} \in R^+ \wedge \vec{J} \notin D_{snk}$ (Equation (4.5b)). For the former condition, the surjective property of a transitive dependence R^+ stated in Lemma 4.4.1 indicates that multiple iterations $\vec{I} \in \text{dom}R^+ \subset D_{snk}$ may depend on the same $\vec{J} \in \text{ran}R^+ \subset D_{snk}$. We thus need to find out distinct iterations $\vec{I} \in \text{dom}R^+$, 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 $\vec{I} \rightarrow \vec{J} \in R^+$. Such iterations $\vec{I}$ can be found by $\text{dom}R^+ - \text{ran}R^+$. 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 $D_{snk}^{ind} \subseteq D_{snk}$ can be computed by taking the union as given in Equations (4.5a) and (4.5b). ■

Consider the PPN in Figure 4.1(a). The sink iterations are described by the domain of process snk, D_{snk} as given in Equation (4.1). Upon computing transitive closure R_{deps}^+ of all dependence relations presented in Section 4.3, there are three transitive dependence relations on D_{snk}, namely R_{13}^+, R_{23}^+, and R_{33}^+. Among them, R_{33}^+ satisfies the condition $\vec{I} \rightarrow \vec{J} \in R^+ \wedge \vec{I} \in D_{snk} \wedge \vec{J} \in D_{snk}$ as stated in Equation (4.5a). The domain and range of R_{33}^+ thus are:

$$\begin{aligned} \text{dom}R_{33}^+ &= \{(i3, j3) \in \mathbb{Z}^2 \mid 1 \leq i3 \leq 7 \wedge 0 \leq j3 \leq 7 - i3\}, \\ \text{ran}R_{33}^+ &= \{(i3, j3) \in \mathbb{Z}^2 \mid 0 \leq i3 \leq 7 \wedge 1 \leq j3 \leq 7 - i3\}. \end{aligned}$$

Then, $\text{dom}R_{33}^+ - \text{ran}R_{33}^+$ in accordance with Equation (4.5a) yields:

$$D_{snk}^{ind1} = \{(i3, j3) \in \mathbb{Z}^2 \mid 1 \leq i3 \leq 7 \wedge j3 = 0\}. \tag{4.6}$$

Furthermore, we compute those iterations that satisfy the second condition in Equation (4.5b) , namely they do not depend on any other iterations in domain D_{snk}. That is:

$$D_{snk}^{ind2} = \{(i3, j3) \mid i3 = 0 \wedge j3 = 0\}. \tag{4.7}$$

Finally, D_{snk}^{ind} can be computed by taking the union of D_{snk}^{ind1} obtained in Equation (4.6) and D_{snk}^{ind2} obtained in Equation (4.7):

$$\begin{aligned} D_{snk}^{ind} &= D_{snk}^{ind1} \cup D_{snk}^{ind2} \\ &= \{(i3, i3) \in \mathbb{Z}^2 \mid 0 \leq i3 \leq 7 \wedge j3 = 0\}. \end{aligned} \tag{4.8}$$

In general, D_{snk}^{ind} computed in accordance with Equation (4.5) 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 $|D_{snk}^{ind}|$. The counting problem can be efficiently solved in polynomial time using the *barvinok* [127] library. Finally, for the PPN shown in Figure 4.1(a) and D_{snk}^{ind} obtained in Equation (4.8), counting the number of integer points in D_{snk}^{ind} yields $|D_{snk}^{ind}| = 8$. This confirms the same number of communication-free partitions, namely 8 as shown in Figure 4.1(b). Also, D_{snk}^{ind} corresponds to the iterations marked by circles show in both Figures 4.1(b) and 4.6(b).

4.5 Communication-free Partitioning Algorithm

If the number of communication-free partitions computed in Section 4.4 is greater than 1, we can transform the initial PPN to a set of communication-free partitions. We first show an example of constructing one of the communication-free partitions for the PPN in Figure 4.1(a). Subsequently, we present the general partitioning algorithm.

An Illustrative Example

Consider the PPN in Figure 4.1(a) and its execution illustrated in Figure 4.1(b). Let us for example construct communication-free partition *Parti. 3* in Figure 4.1(b). In the partitioning algorithm for this example, 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 $(i3, j3) = (3, 0) \in D_{snk}^{ind}$ of process P_3 as computed in Equation (4.8). All transitive dependence relations R_{33}^+, R_{23}^+, and R_{13}^+ to iteration $(3, 0)$ are computed in Equations (4.3a) to (4.3c) and illustrated in Figure 4.6(b). In the first step of the partitioning algorithm for *Parti. 3*, we instantiate process P_{33} (see Figure 4.7) of PPN process P_3 through R_{33}^+. Process P_{33} performs the same computational function as the original PPN process P_3 does. The only difference is that process P_{33} only executes in a subdomain D_{33} of the original domain D_3. For *Parti. 3*, besides that iteration $(3, 0)$ belongs to domain D_{33}

of process P_{33}, P_{33} contains also iterations $(2,1)$, $(1,2)$, and $(0,3)$ of P_3, on which iteration $(3,0)$ depends, as shown in Figure 4.6(b). These iterations can be derived by "substituting" iteration $(3,0)$ in R^+_{33} (see Equation (4.3c)), denoted as $R^+_{33}((3,0))$:

$$\begin{aligned} R^+_{33}((3,0)) &= \{(i3',j3') \mid (3,0) \rightarrow (i3',j3') \in R^+_{33}\} \\ &= \{(i3',j3') \mid 0 \leq i3' \leq 2 \wedge j3' = 3 - i3'\}. \end{aligned} \tag{4.9}$$

Then, domain D_{33} for *Parti. 3* can be obtained by taking a union of iteration $(3,0)$ with the ones computed in Equation (4.9):

$$\begin{aligned} D_{33} &= (3,0) \cup R^+_{33}((3,0)) \\ &= \{(i3',j3') \mid 0 \leq i3' \leq 3 \wedge j3' = 3 - i3'\}. \end{aligned} \tag{4.10}$$

Second, a process P_{23} (see Figure 4.7) of PPN process P_2 is instantiated due to R^+_{23} for *Parti. 3*. Domain D_{23} contains iteration (3) of P_2 as shown Figure 4.6(b). It can be derived by "substituting" domain D_{33}, obtained in Equation (4.10), in R^+_{23} (see Equation (4.3b)), denoted as $R^+_{23}(D_{33})$:

$$\begin{aligned} D_{22} = R^+_{23}(D_{33}) = &\ \{(j2) \mid (i3,j3) \rightarrow (j2) \in R^+_{23} \wedge (i3,j3) \in D_{33}\} \\ = &\ \{(i2) \in \mathbb{Z} \mid i2 = 3\}. \end{aligned} \tag{4.11}$$

Finally, we need to instantiate a process P_{13} (see Figure 4.7) with domain D_{13} due to R^+_{13}. Domain D_{11} corresponds to iteration (3) in domain D_{P1} as shown in Figure 4.6(b). Analogous to obtaining domain D_{23}, domain D_{13} can be obtained by "substituting" domain D_{33} in R^+_{13} (see Equation (4.3a)):

$$D_{13} = R^+_{13}(D_{33}) = \{(i1) \in \mathbb{Z} \mid i1 = 3\}.$$

Once all processes for *Parti 3* are instantiated, next we instantiate edges for the new processes. Basically, if an edge in the initial PPN is incident with the new process, a new edge is instantiated. For *Parti. 3*, edge E_3 in the initial PPN is incident with the new process P_{33}. Then, a new edge E_{33} is instantiated with the associated input port domain $D_{IP_{33}}$, output port domain $D_{OP_{33}}$, and dependence relation R_{33}:

$$\begin{aligned} D_{IP_{33}} = &\ D_{IP_3} \cap D_{33} \\ = &\ \{(i3,j3) \mid 1 \leq i3 \leq 3 \wedge j3 = 3 - i3\}, \\ D_{OP_{33}} = &\ D_{OP_3} \cap D_{33} \\ = &\ \{(i3',j3') \mid 0 \leq i3' \leq 2 \wedge j3' = 3 - i3'\}, \\ R_{33} = &\ \{(i3,j3) \rightarrow (i3',j3') \mid (i3,j3) \in D_{IP_{33}} \wedge (i3',j3') \in D_{OP_{33}} \\ &\ \wedge i3' = i3 - 1 \wedge j3' = j3 + 1\}. \end{aligned} \tag{4.12}$$

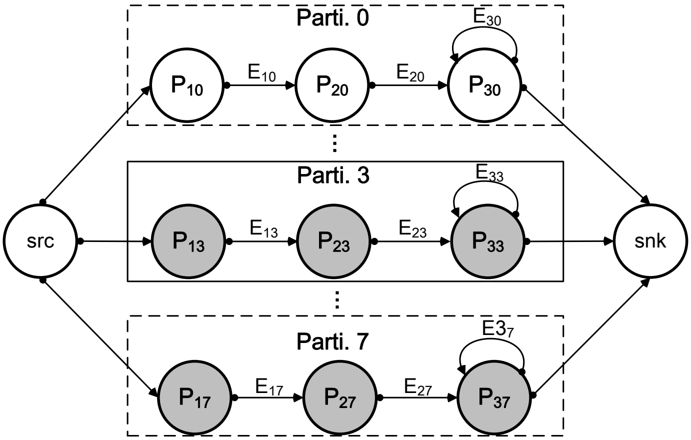


Figure 4.7: The PPN in Figure 4.1(a) after communication-free partitioning.

Two other edges E_{13}, E_{23} can be instantiated in a similar way, due to edges E_1, E_2 in the initial PPN. In this way, communication-free partition *Parti. 3* shown in Figure 4.1(b) is constructed and illustrated by the solid box in Figure 4.7. In the next step, we merge all process instances P_{33}, P_{23}, and P_{13} into a single compound process *Parti. 3* as shown in Figure 4.4(a). We generate a static schedule, similar to the one proposed in [124], that executes all dependent iterations of the new processes as close as possible.

General Partitioning Algorithm

In general, to instantiate new processes and edges, we devise Algorithm 3. The input to Algorithm 3 is a PPN with all transitive dependences ($\mathcal{E}^+$) computed in Section 4.3 and $D_{snk}^{ind} \subseteq D_{snk}$ obtained in Theorem 4.4.1. Every sink iteration $\vec{K} \in D_{snk}^{ind}$ is used to identify a distinct communication-free partition. The output of Algorithm 3 is $|D_{snk}^{ind}|$ communication-free partitions. The core part of the algorithm is presented below.

Algorithm 3 starts partitioning a PPN from the sink process, namely partitioning P_{snk} into $|D_{snk}^{ind}|$ number of processes P_{snk_inst}. For each iteration $\vec{K} \in D_{snk}^{ind}$ of the sink process, we instantiate a new process P_{snk_inst}. The loop iterates over all PPN processes to instantiate all processes in all communication-free partitions. Basically, for a particular partition, we construct the domain for each new process through all transitive dependence relations R_E^+ on iteration $\vec{K}$. First, we construct domain $D_{P_{snk\_inst}}$. If this iteration $\vec{K}$ transitively depends on other iterations in domain D_{snk},

Algorithm 3: Communication-free partitioning procedure

```
Input: A $PPN = (\mathcal{P}, \mathcal{E})$, $\mathcal{E}^+$, and $D_{snk}^{ind}$ obtained in Theorem 4.4.1.
Result: A $PPN' = (\mathcal{P}', \mathcal{E}')$.
$\mathcal{P}' \leftarrow \emptyset$, $\mathcal{E}' \leftarrow \emptyset$ ;
Get sink process $P_{snk}$, $D_{snk\_inst} \leftarrow \emptyset$ ;
foreach $\vec{K} \in D_{snk}^{ind}$ do
    $P_{snk\_inst} \leftarrow P_{\text{snk}}$ ;
    foreach $Edge\ E^+ \in \mathcal{E}^+$ incident with $P_{snk}$ do
        Get $R_E^+$ associated with edge $E^+$ ;
        if $\vec{K} \notin \text{dom} R_E^+$ then
            continue;
        if $\text{ran} R_E^+ \subseteq D_{snk}$ then /* $\vec{K}$ depends on other iterations in $D_{snk}$ */
            $D_{P_{snk\_inst}} \leftarrow D_{P_{snk\_inst}} \cup \vec{K} \cup R_E^+(\vec{K})$ ;
        else /* $\vec{K}$ depends on another process $P$ */
            $D_{P_{snk\_inst}} \leftarrow D_{P_{snk\_inst}} \cup \vec{K}$ ;
            Get process $P \in \mathcal{P}$ incident with edge $E^+$;
            $P_{inst} \leftarrow P$ ;
            $D_{P_{inst}} \leftarrow R_E^+(D_{snk\_inst})$ ;
            $\mathcal{P}' \leftarrow \mathcal{P}' \cup P_{inst}$ ;
        $\mathcal{P}' \leftarrow \mathcal{P}' \cup P_{snk\_inst}$ ;
foreach $P_{inst} \in \mathcal{P}'$ do
    $\mathcal{E}_{inst} \leftarrow instantiateChannels(P_{inst}, \mathcal{E})$ ;
    $\mathcal{E}' \leftarrow \mathcal{E}' \cup \mathcal{E}_{inst}$;
```

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 (4.9)). Therefore, in this case, we can obtain $D_{P_{snk\_inst}}$ in Algorithm 3. 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 $\vec{K}$. Also,

Algorithm 4: Procedure *instantiateChannels*

```
Input: A process instance $P_{inst}$ and a set edges $\mathcal{E}$.
Result: A set of edges $\mathcal{E}_{inst}$ incident with process instance $P_{inst}$.
Get $D_{P_{inst}}$ of $P_{inst}$;
foreach Channel $E \in \mathcal{E}$ incident with $P_{inst}$ do
    Get $D_{IP}$ and $D_{OP}$ associated with edge $E$;
    $E_{inst} \leftarrow E$ ;
    if $D_{IP} \cap D_{P_{inst}} \neq \emptyset$ and $D_{OP} \cap D_{P_{inst}} \neq \emptyset$ then /* a self-edge */
        $D_{IP\_inst} \leftarrow D_{inst} \cap D_{IP}$ , dom$R_{E_{inst}} \leftarrow D_{IP\_inst}$;
        $D_{OP\_inst} \leftarrow D_{inst} \cap D_{OP}$, ran$R_{E_{inst}} \leftarrow D_{OP\_inst}$ ;
        $\mathcal{E}_{inst} \leftarrow \mathcal{E}_{inst} \cup E_{inst}$ ;
    else if $D_{IP} \cap D_{P_{inst}} \neq \emptyset$ and $D_{OP} \cap D_{P_{inst}} = \emptyset$ then /* an incoming
    edge */
        $D_{IP\_inst} \leftarrow D_{inst} \cap D_{IP}$, dom$R_{E_{inst}} \leftarrow D_{IP\_inst}$ ;
        $\mathcal{E}_{inst} \leftarrow \mathcal{E}_{inst} \cup E_{inst}$ ;
    else if $D_{IP} \cap D_{P_{inst}} = \emptyset$ and $D_{OP} \cap D_{P_{inst}} \neq \emptyset$ then /* an outgoing
    edge */
        $D_{OP\_inst} \leftarrow D_{inst} \cap D_{OP}$, ran$R_{E_{inst}} \leftarrow D_{OP\_inst}$ ;
```

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. 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 given as:

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

An example of the applying operation can be seen in Equation (4.11). Finally, all process instances in the same communication-free partitions can be instantiated.

Once all processes for each communication-free partition are instantiated, as the next step, we need to instantiate edges for all processes in Algorithm 3. The procedure of instantiating all edges for a process instance is depicted in Algorithm 4. As input, it takes a process P_{inst} with constructed domain D_{inst} and all edges $\mathcal{E}$ in the initial PPN. The algorithm outputs a set of edges $\mathcal{E}_{inst}$ incident with process instance P_{inst}. In Algorithm 4, if both the input port and output port of a edge E are incident with P_{inst}, a new self-edge E_{inst} is instantiated with the corresponding input and output port domains. An example of instantiating self-edge E_{33} for new process

P_{33} can be seen in Equation (4.12). If only the input port or output port of an edge E is incident with P_{inst}, it represents a dependence relation from/to another process instance in the same communication-free partition. In other words, it is either an incoming or outgoing edge of process instance P_{inst}. In this case, we instantiate only one edge with its corresponding input and output port domains. Therefore, using Algorithm 4, we can instantiate all edges incident with a new process P_{inst}.

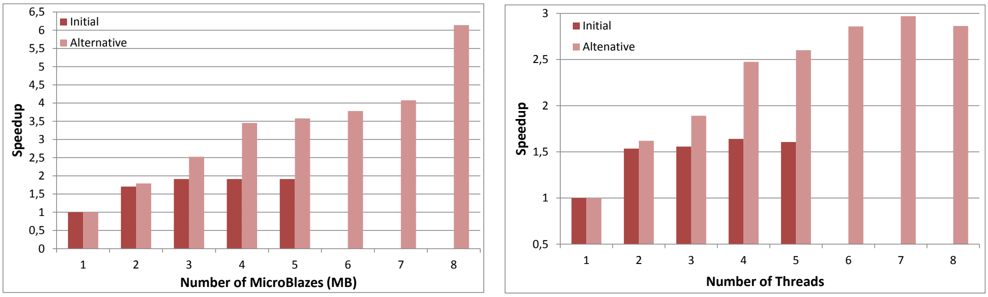
4.6 Experimental Results

In this section, we present the performance results obtained by applying our approach explained previously 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 implemented the partitioning algorithm presented in Section 4.5 in PNtool as part of the Daedalus^{RT} design flow shown in Figure 1.7 on page 13. We conducted all experiments using the ESPAM [96] tool, the Xilinx Platform Studio 13.2, and Microsoft Visual Studio 2008. All generated programs were compiled using compilers `mb-g++4.6.2` and `g++4.52` on the selected platforms respectively, with optimization level `-O2`.

Case Studies

We considered two real-life applications modeled using the PPN MoC, namely a Motion-JPEG (MJPEG) encoder used in [34] and the FM radio application taken from the StreamIT benchmark suite [54]. The MJPEG encoder encodes frames of 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



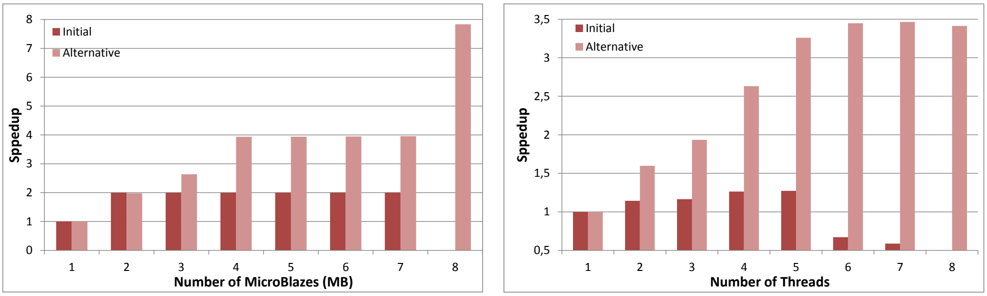
(a) (b)

Figure 4.8: Performance results of mapping the MJPEG encoder onto (a) FPGA-based MPSoC platforms and onto (b) a desktop multi-core platform.

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 4.8(a) for the FPGA platform and in Figure 4.8(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 4.8(a) and 4.8(b)) leads to better per-



(a) (b)

Figure 4.9: Performance results of mapping the FM radio application onto (a) FPGA-based MPSoC platforms and onto (b) a desktop multi-core platform.

formance. Our approach outperforms the mapping of the initial PPN by 5% to 87.05% on 2 to 5 MBs. As shown in Figure 4.8(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 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 [54] 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 6.14X 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 4.8(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, 2.97X 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 4.9(a) for the FPGA platform and in Figure 4.9(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, 1.99X speedup is achieved on 2 MBs. On the desktop platform shown in Figure 4.9(b), the best mapping of the initial PPN is found using 5 threads occupying 4 cores, i.e., 1.27X 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 communication from the data source and to the data sink. For the results on the FPGA platform shown in Figure 4.9(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 4.9(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.

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 [101]. Other benchmarks are less complex than the benchmarks listed in Table 4.1 in terms of their characteristics. The characteristics of each benchmark are given in columns 2 to 4 in Table 4.1. The benchmarks differ in the of number of PPN processes (denoted by $|\mathcal{P}|$) and edges (denoted by $|\mathcal{E}|$) in the initial PPNs, as well as dimensions of data arrays accessed in

Table 4.1: Execution time on benchmarks.

Benchmark	$\lvert\mathcal{P}\rvert$	$\lvert\mathcal{E}\rvert$	Array dimensions	Execution time (sec.)
ADI[1]	12	67	3	2.644
Gram-schmidt	8	19	2	0.924
FDTD[2]	9	27	2	0.604
Correlation	12	20	2	0.076
Reg-detect[3]	8	11	3	0.068
Dynprog[4]	8	12	3	0.064
Gauss[5]	11	18	2	0.044
Covariance	8	11	2	0.032

[1] ADI: Alternating direction implicit solver

[2] FDTD: 2D finite difference time domain kernel

[3] Reg-detect: Regularity detection

[4] Dynprog: Dynamic programming (2D)

[5] Gauss: 2D gauss blur filter for image processing

PPN processes. For instance, the ADI solver in Table 4.1 operates on 3 dimensional data arrays. In practice, it can be seen that, from the last column in Table 4.1, 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 benchmark.