Transformations for polyhedral process networks

Meijer, S.

Chapter 5

Appling Transformations in Combination

In Chapter 3 we have discussed a compile-time approach for evaluating the *process splitting* transformation [51, 78, 79], and in Chapter 4 an approach for evaluating the *process merging* transformation [53]. These two parameterized transformations play a vital role in meeting the performance/resource constraints. The splitting transformation is parameterized in the sense that a given process can be split up in many different ways, and the designer must choose a specific splitting factor (i.e., the number of created copies). For the merging transformation, it is obvious that the designer must decide which processes to merge. The problem is that, for both transformations, the designer must select a particular process(es) to apply the transformations on in order to achieve good results. This is not a straightforward task as we explain in Section 5.2.2. In addition to this, both transformations can be applied one after the other and in a different order with different parameters which may, or may not, give better results than applying one transformation only. Therefore, in this chapter we

- investigate whether applying the two transformations in combination can give better performance results than applying only one,
- propose a solution approach that solves the very difficult problem of determining the best order of applying the transformations and the best transformation parameters,
- relieve the designer from the difficult task of selecting processes on which the applied transformations have the largest positive performance impact, and
- present a solution approach that exploits available data-level parallelism in cyclic PPNs and/or PPNs with stateful processes.

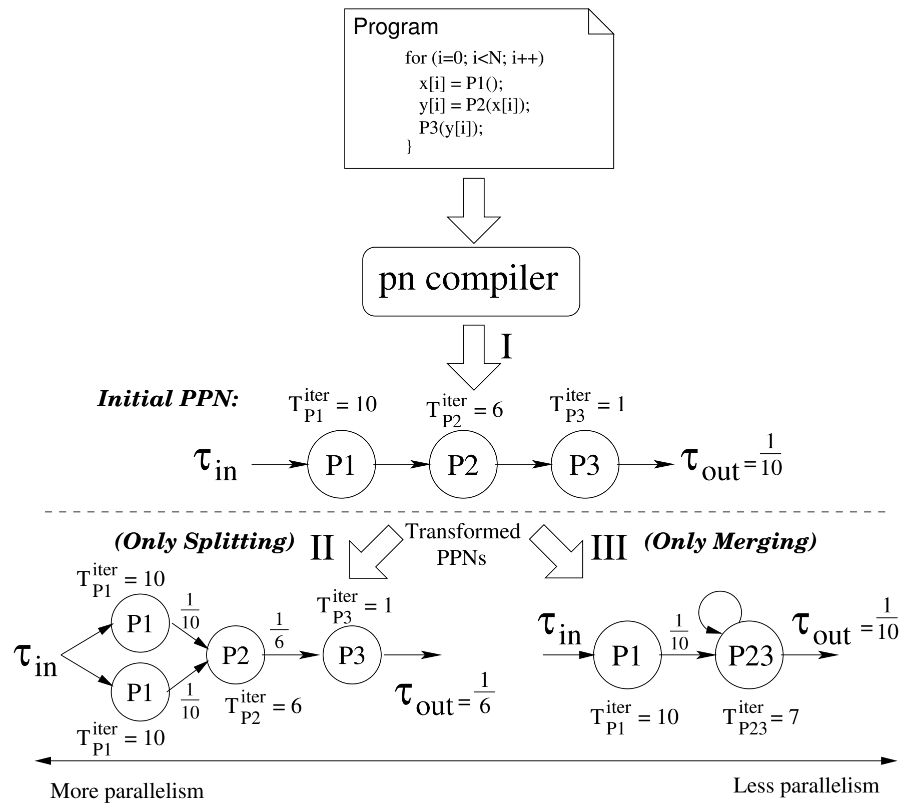


Figure 5.1: Deriving and Transforming Process Networks

In this Chapter, we apply the different transformations in combination on the initial PPN shown in Figure 5.1. Arrow `II` is an example of applying the process splitting transformation on process *P1*. The transformed network has two processes *P1* executing the same function such that the data tokens are delivered twice faster to the consumer process *P2*. Recall from Chapter 3, that we refer to the two processes *P1* as process partitions of *P1*. Arrow `III` is an example of transforming the initial PPN by applying the merging transformation on processes *P2* and *P3* to create compound process *P23*. The problem how to apply each transformation has been discussed in the previous chapters. However, still a remaining challenge is to devise a holistic approach to help the designer in transforming and mapping PPNs onto the available processing elements of the provided target platform to achieve even better performance results using the two transformations in combination. In the next section, we first investigate the effects on the performance results of applying both transformations in combination. Next, we propose a solution how to order them, and finally we present two case-studies.

5.1 Impact of the Transformation on Performance Results

We investigate whether applying both the process splitting and merging transformations in combination gives better performance results than applying only one transformation. Consider the initial and transformed PPNs in Figure 5.1. Each process P_i is annotated with the time $T_{P_i}^{iter}$ that is required to execute one process iteration, which includes the time for executing the process function and also the FIFO communication costs, see Definition 3.9. For example, a process iteration of $P1$ is completed in 10 time units, i.e., $T_{P1}^{iter} = 10$, while $P2$ is a computationally less intensive process as one process iterations is completed in only 6 time units, i.e., $T_{P2}^{iter} = 6$. Process $P1$ determines therefore the system throughput of the initial PPN. The throughput is denoted by τ_{out} and we define it as the number of tokens produced by the network per time unit (see Definition 18 in Section 4.2). Since $P1$ is the most computationally intensive process that fires each 10 time units, the throughput and number of produced tokens is $\frac{1}{10}$ tokens per time unit. Now we show and discuss many different examples in this section to illustrate how difficult it is for a designer to apply transformation, even for such a simple initial PPN as shown in Figure 5.1.

5.1.1 Transforming a PPN to Create More Processes

If we want to increase the performance results for a given PPN, the number of processes can be increased using the process splitting transformation to benefit from more parallelism. In this subsection we, therefore, show two different PPNs consisting of 4 processes that are derived from the same initial PPN consisting of 3 processes. The first transformed PPN is derived from the initial PPN in Figure 5.1 using only the process splitting transformation, and the second is derived from the initial PPN using both the process splitting and merging transformation.

Transformed $PPN1$ (only splitting)

We split up process $P1$ two times as shown in Figure 5.1. Then there are 2 processes that generate data in parallel for consumer process $P2$. As a result, process $P2$ receives its input data twice as fast. Therefore, we say that process $P2$ receives its data with an aggregated throughput of $\frac{1}{10} + \frac{1}{10} = \frac{1}{5}$. We know that the slowest process in a network determines the system throughput and to check this, we compare the incoming throughput of a process with the time it takes to fire that process function. While $P2$ receives its input data with a throughput of $\frac{1}{5}$ tokens per time unit, it can only produce tokens with a throughput of $\frac{1}{6}$ ($T_{P2}^{iter} = 6$). This means that the input tokens arrive faster than $P2$ can process them. To calculate the overall system throughput, we therefore propagate the throughput $\tau = \frac{1}{6}$ of $P2$ to sink process $P3$ and compare what is slower: the arrival of the input data, or the firing of process $P3$.

We see that *P3* can process data much faster than it actually receives since $T_{P3}^{iter} = 1$, but still it produces tokens with a throughput of $\frac{1}{6}$ caused by the slowest process *P2*. The overall system throughput is therefore $\tau_{out} = \frac{1}{6}$ and is determined by *P2*. Thus, we have derived a PPN that gives a throughput $\tau_{out} = \frac{1}{6}$ that is much better than the initial throughput $\tau_{out} = \frac{1}{10}$.

Now we investigate whether we can derive another network with 4 processes, using both the process splitting and merging transformations in combination, that gives even better performance results than our previous example.

Transformed *PPN2* (splitting+merging)

We apply first the process splitting transformation on processes *P1*, *P2*, and *P3* from the initial PPN in Figure 5.1 to derive the transformed PPN shown in Figure 5.2 A). Two independent data paths are created each consisting of 3 processes.

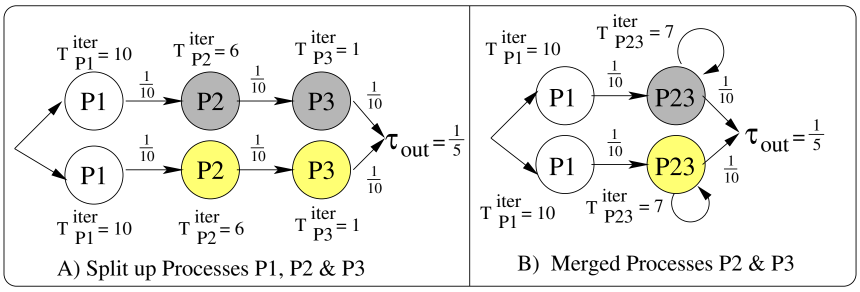


Figure 5.2: Transformed PPN2: Splitting and Merging to Create 4 Processes

In each data path, process *P1* is the bottleneck process such that tokens are delivered with a throughput of $\frac{1}{10}$. Since there are two data paths, we say that the overall system throughput of the transformed PPN in Figure 5.2 A) is $\tau_{out} = \frac{1}{5}$. When we merge *P2* with *P3*, process *P1* remains the bottleneck and the throughput is unaffected as shown in Figure 5.2 B). Thus, we have derived a PPN with 4 processes that gives better performance results compared to the previous example *Transformed PPN1 (only splitting)* shown in Figure 5.1. That is, applying both transformations in combination achieves a throughput of $\tau_{out} = \frac{1}{5}$, while applying only the process splitting transformation gives a throughput $\tau_{out} = \frac{1}{6}$. In fact, to create a PPN with n processes from the initial PPN in Figure 5.1, the best performance results that can be achieved by using the process splitting transformation only, will never be better than the best performance results that can be achieved by applying both transformations in combination. Therefore, this example shows that both transformations must be used in combination to achieve better performance results.

5.1.2 Transforming a PPN to Reduce the Number of Processes

A designer sometimes needs to reduce the number of processes for a given PPN in order to meet resource constraints. Another reason to merge processes, is that in some cases the same performance can be achieved using less processes. In this subsection, our objective is to derive a PPN consisting of 2 processes when this is required for one of the two reasons mentioned above. We start with the initial PPN in Figure 5.1 that has 3 processes and investigate again whether the combination of applying the transformations is important when the number of processes in the network must be reduced.

Transformed $PPN3$ (only merging)

A transformed PPN with 2 processes is shown in the bottom right part of Figure 5.1, which is obtained by applying only the process merging transformation. The resulting network has the same throughput as the initial PPN, but uses one process less. By merging 2 light-weight processes $P2$ and $P3$, process $P1$ remains the most computationally intensive process. As a result, the system throughput remains the same as in the initial network, i.e., $\tau_{out} = \frac{1}{10}$.

Transformed $PPN4$ (splitting+merging)

An alternative using both the process splitting and merging transformations is shown in Figure 5.3.

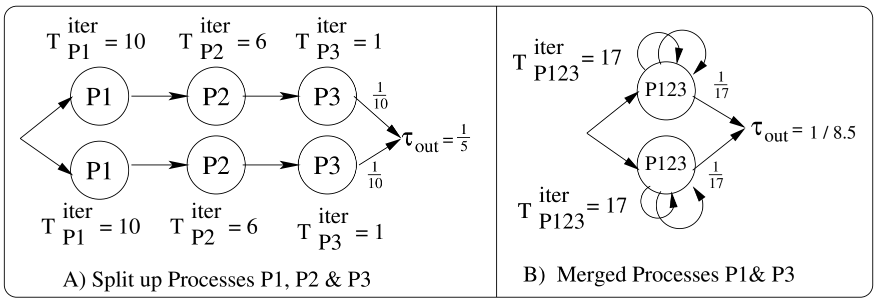


Figure 5.3: Transformed PPN4: Creating 2 Load-Balanced Tasks

All processes are first split up two times as shown in Figure 5.3 A). Then, two compound processes are created by merging a process partition of each process into a compound process $P123$ as shown in Figure 5.3 B). The time for one process iteration of the compound process is $T^{iter}_{P123} = T^{iter}_{P1} + T^{iter}_{P2} + T^{iter}_{P3} = 17$ time units, because all process functions are executed sequentially. This means that the compound process delivers tokens with a throughput of $\tau_{P123} = \frac{1}{17}$. Since we have 2 compound processes, the resulting overall throughput is $\tau_{out} = \frac{1}{8.5}$, which is better than the throughput $\tau_{out} = \frac{1}{10}$ of our previous example *Transformed PPN3 (only merging)* shown in Figure 5.1. This is another example which shows that both transformations should be applied in combination to obtain better performance results, which cannot be obtained by only one transformation (i.e., the merging transformation in this case).

5.1.3 The Optimization Pitfall: Performance Degradation

We have shown that there is great potential in using both transformations in combination, but a designer should be very careful how the transformations are applied, otherwise performance degradation may be encountered. We illustrate this with two examples using both the process splitting and merging transformations. First we show an example for a PPN with 4 process and then for a PPN with 2 processes.

Transformed *PPN5* (splitting+merging)

We start with the initial PPN in Figure 5.1, which consists of 3 processes, and split up both processes *P1* and *P2* to obtain the PPN shown in Figure 5.4 A).

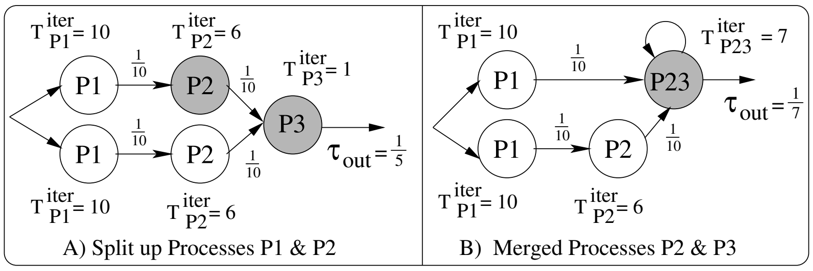


Figure 5.4: Transformed PPN5: Splitting and Merging to Create 4 Processes

The network has a throughput of $\frac{1}{5}$ using 5 processes, while our objective is to use 4 processes. Therefore, we merge two light-weight processes *P2* and *P3*. The created compound process *P23* has a process iteration time $T^{iter}_{P23} = 7$ time units and is the bottleneck process of the network. The overall system throughput is, therefore, determined by *P23* and is $\tau_{out} = \frac{1}{7}$. In this way, we have derived another PPN with 4 processes that performs better than the initial process network ($\tau_{out} = \frac{1}{10}$). However, the throughput $\tau_{out} = \frac{1}{7}$ is worse than the throughput achieved by applying only the splitting transformation, i.e., *transformed PPN1 (only splitting)* in Figure 5.1 with a throughput of $\tau_{out} = \frac{1}{6}$ and subsequently also worse than *Transformed PPN2* shown in Figure 5.2 B) that has a throughput $\tau_{out} = \frac{1}{5}$.

Transformed *PPN6* (splitting+merging)

We have shown two examples to transform the initial PPN in Figure 5.1 into a PPN with 2 processes in *Transformed PPN3* and *Transformed PPN4*. Both give good performance results, but now we give an example of a PPN that performs worse. Another possibility to create a PPN with 2 process is to first split up the computationally most intensive process *P1* as shown in Figure 5.5 A). Then, two compound processes

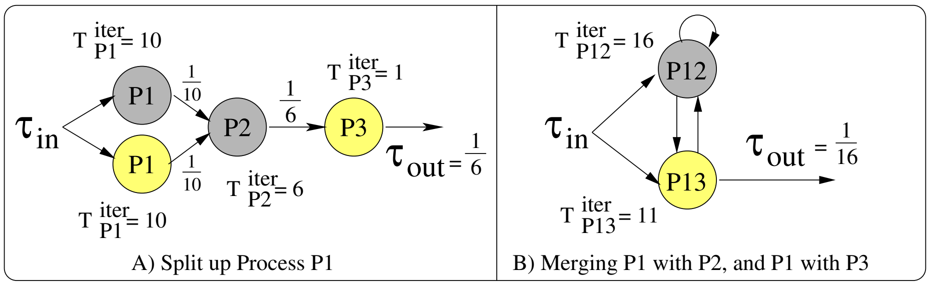


Figure 5.5: Transformed PPN6: Splitting and Merging to Create 2 Processes

are created, one by merging process *P1* with *P3*, and the other one by merging process *P1* and *P2*. We see that a topological cycle is introduced by merging processes in this way and we find that the system throughput is $\tau_{out} = \frac{1}{16}$ tokens per time unit. This result is worse than *Transformed PPN3* and *Transformed PPN4* that have a throughput of $\tau_{out} = \frac{1}{10}$ and $\tau_{out} = \frac{1}{8.5}$, respectively.

In this section, we have shown that it is necessary to apply both the process splitting and merging transformations in combination to achieve better performance results that cannot be achieved by applying only one transformation in isolation. On the other hand, performance degradation may be encountered if the transformations are not applied properly. So the question is how a designer should apply the transformations properly, i.e., choosing the best possible order of transformations and their parameters. In the next section, we show our solution approach that addresses these issues.

5.2 Compile-Time Solution for Transformation Ordering

Before introducing our solution in a more formal way, we show how our approach intuitively works for the examples discussed in Section 5.1. We have already shown 3 different PPNs consisting of 4 processes that were derived from the same initial PPN. The first transformed PPN is obtained by using only the splitting transformation as shown in Figure 5.1. In two other examples, shown in Figure 5.2 B) and

Figure 5.4 B), different networks were obtained by consecutively using the process splitting and merging transformations. Our solution approach, however, gives a different solution and also gives better performance results as we show with the examples in Figure 5.6.

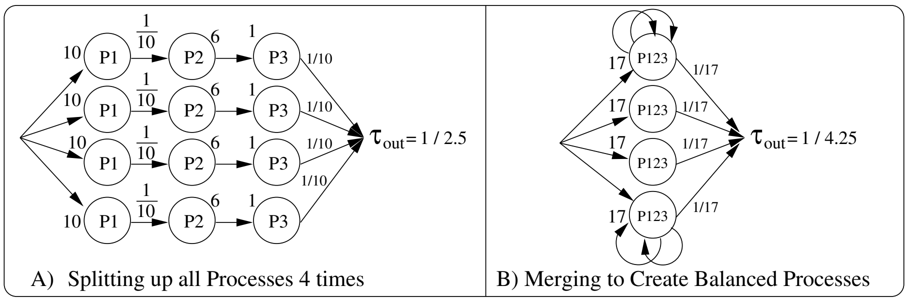


Figure 5.6: Creating 4 Load-Balanced Tasks

In our simple, elegant but yet very effective solution approach, we first split up all processes with a splitting factor that is specified by the designer. This splitting factor can, for example, be the number of available processing elements of the target platform, or simply the number of tasks the designer wants to create. Since in our examples the goal is to transform and create a PPN with 4 processes, we split up all processes 4 times as shown in Figure 5.6 A). In this way, we create a PPN consisting of 12 processes. Next, we merge back process partitions into compound processes such that they contain one process partition of each process. Figure 5.6 B) shows these compound processes $P123$. Note that the self-edges for two compound processes have been omitted for the sake of clarity. The time to execute one process iteration of the compound processes is 17 time units, which is obtained by summing the process iteration time of the individual processes. Thus, we know that each compound process produces $\frac{1}{17}$ tokens per time unit. Since there are 4 compound processes, the overall system throughput $\tau_{out} = \frac{4}{17} = \frac{1}{4.25}$, which is better than all other transformed PPNs with 4 processes shown in Figure 5.1, Figure 5.2 B), and Figure 5.4 B).

The initial PPN in Figure 5.1 is transformed in a similar way if the number of processes needs to be reduced. We have already shown 2 examples and our solution is already given in Figure 5.3; all processes are first split up 2 times, and then compound processes are created by merging different process partitions such that the resulting transformed network consists of 2 processes.

5.2.1 Creating Load-Balanced Tasks

While we illustrated our solution approach with examples in the previous section, a more formal description of our solution approach is given with the pseudo-code in Algorithm 2. We create a number of tasks from an initial PPN based on the combination of two transformations: *i)* the processes are split-up first, and *ii)* load-balanced tasks are created by using the process merging transformation.

Algorithm 2 : Task Creation Pseudo-code

```
Require: A Polyhedral Process Network $PPN$ with $n$ processes,
Require: A process splitting factor $u$.
  for all $P_i \in PPN$ do
    $\{P_{i1}, P_{i2}, .., P_{iu}\} = split(P_i, u)$
  end for
  for $i = 1$ to $u$ do
    $P_{Ci} = merge(\{P_{1i}, P_{2i}, .., P_{ni}\})$
  end for
  return all compound processes $P_{Ci}$
```

Algorithm 2 uses two functions: `split` and `merge`. For the former, we refer to Chapter 3 in which it is shown that a process can be split up in many different ways and how to select the best splitting. We use the approach in Chapter 3 to select and perform the processes splitting. For the process merging transformation, we rely on the approach described in Chapter 4. We add to this approach a procedure to cluster producer-consumer pairs of processes. By clustering producer-consumer processes, communication between these processes stays within one compound process after merging. Thus, it tries to avoid communication and synchronization of different compound processes. An example of this is given in Figure 5.6. One process partition of *P1* has only one channel to *P2*, which in turn has only one channel to *P3*. Merging processes in this sequence results in compound processes that do not have any communication channels among them. It is not always possible to obtain completely independent compound processes. If one producer process has multiple channels to consumer processes, as shown in Figure 5.7 A), one particular consumer has to be selected and merged with the producer.

If we start with the first partition of *P1*, i.e., grey process *P1* in Figure 5.7 A), then we see that it has two outgoing channels to two process partitions of *P2*. Regardless which partition of *P2* is chosen for merging, the resulting compound processes will have channels for data communication between them as shown in Figure 5.7 B). In our approach, we simply consider the first outgoing channel and corresponding consumer process, and merge it with the producer. We mark this consumer as being merged already, to avoid that it will be selected again.

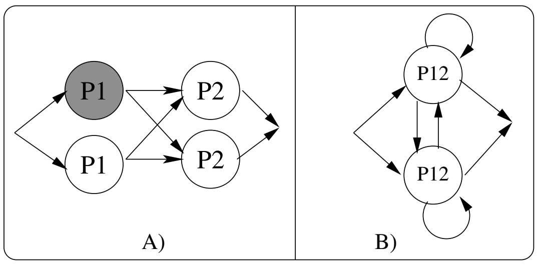


Figure 5.7: Different Merging Options

5.2.2 Selecting Processes for Transformations

Our solution approach in Section 5.2.1 solves another problem indicated in the introduction of this chapter, i.e., how to select processes to be transformed on which the transformations have the largest positive performance impact. For the process splitting, it is important to find the bottleneck process of the network, because splitting is the most beneficial when applied on the bottleneck process. For process merging, it is important to avoid merging the bottleneck process, i.e., not introducing an even larger bottleneck process. In general, however, it is not possible at all to determine a single bottleneck process. The reason is that, in PPNs, different data paths can transfer a different number of tokens. As a result, different processes can determine the overall system throughput at different stages during the execution of the network, which we illustrate with the example shown in Figure 5.8.

The network has two datapaths $DP1 = (P1, P2, P3, P6)$ and $DP2 = (P1, P4, P5, P6)$ that transfer a different number of tokens. This is the result of the communication patterns `[1100000]` and `[0011111]` at which process $P1$ writes to its outgoing FIFO channels. A ”`1`” in these patterns indicates that data is read/written and a ”`0`” that no data is read/written. So, the FIFO channel connecting $P1$ and $P2$, for example, is written the first two firings of $P1$, but not in the remaining 5 firings. As a consequence of these patterns, more tokens are communicated through the second datapath $DP2$. At the bottom of Figure 5.8, the different time lines of the processes are shown. Each block corresponds to a firing of that process producing data, and the arrow indicates the dependent consumer process. In this way, a full simulation of the process network is shown. We observe that, despite process $P2$'s largest process iteration time $T^{iter}_{P2} = 10$ time units, process $P4$ with $T^{iter}_{P4} = 6$ is determining the throughput most of the time. This illustrates that, in general, due to the varying and possibly complicated communication patterns, it is not possible to decide which process to split up for a more balanced network. Our solution approach in Section 5.2.1, solves this problem as the transformations are applied on all

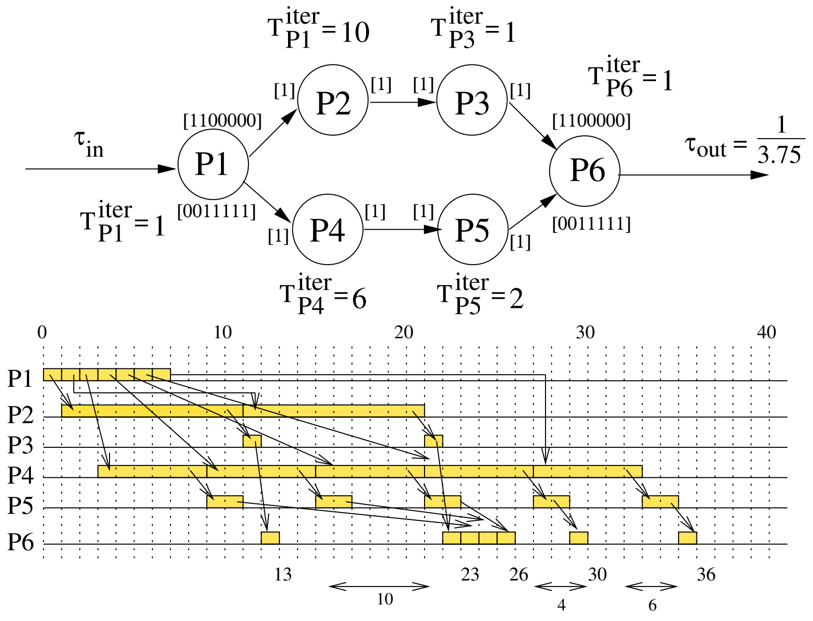


Figure 5.8: What is the Bottleneck Process: $P2$ or $P4$?

processes and, therefore, it is not necessary to select particular processes.

5.3 Exploiting Data-Level Parallelism

The idea of our approach presented in Section 5.2 is to create load-balanced tasks that exploit *data-level parallelism* as much as possible. In this section, we want to show that our simple solution always results in performance gains when there is data-level parallelism to be exploited. The degree of data-level parallelism that can be exploited is determined by:

1. Processes with *self-edges* in a PPN. Similar to the definition used in [31], we refer to data-level parallelism when processes do not dependent on previous firings of itself. Obviously, when there is no self-edge, the process is *stateless* and an arbitrary number of independent process partitions can be created that run in parallel. When a process has a self-edge, however, it produces data for itself and there exists a dependency between different firings of that process. Then, we refer to such a process as *stateful*.

2. *Cycles* in a PPN. A cycle can be responsible for sequential execution of the processes involved in the cycle. If this is the case, we call it a *true cycle*.

Despite stateful processes and topological cycles, PPNs may still reveal some data-level parallelism which is exploited by our solution approach. This means that our solution approach gives better performance results when there is data parallelism to be exploited, and the same performance as the initial PPN if there is nothing to be exploited. In addition to cycles and stateful process, the workload balancing of the initial PPN is another important factor that determines whether performance gains are possible. We therefore first discuss this workload balancing before we elaborate how to exploit more data-level parallelism for stateful processes and cyclic PPNs.

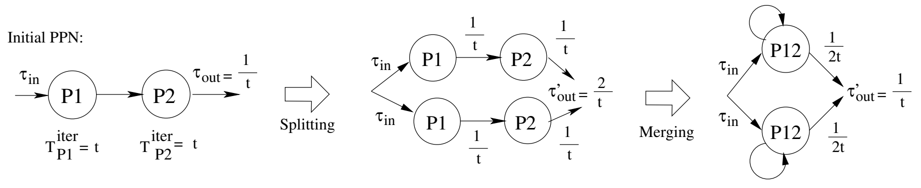


Figure 5.9: Simple Acyclic Producer/Consumer

Balanced PPNs

Let us consider the PPN shown in Figure 5.9 and its two processes $P1$ and $P2$.

- The PPN and its processes $P1$ and $P2$ shown in Figure 5.9 are balanced, because $T_{P1}^{iter} = T_{P2}^{iter} = t$ time units. The throughput of the PPN is therefore $\tau_{out} = \frac{1}{t}$. If we apply splitting and merging, as illustrated with the arrows in Figure 5.9, then a compound process has a throughput of $\tau = \frac{1}{2t}$. Since there are two compound processes the overall throughput is $\tau'_{out} = 2 \cdot \frac{1}{2t} = \frac{1}{t}$. Thus, we see that the new throughput τ'_{out} is the same as the throughput of the initial PPN, that is, $\tau'_{out} = \tau_{out}$.

Now let us consider the other case:

- Suppose that the PPN in Figure 5.9 and its processes $P1$ and $P2$ are imbalanced, then we have $T_{P1}^{iter} = t$ and $T_{P2}^{iter} = t + x$, where $x > 0$. The throughput of the initial PPN is $\tau_{out} = \frac{1}{t+x}$. Then, we apply our solution approach and create 2 independent streams. Each compound process has a throughput of $\tau = \frac{1}{T_{P1}^{iter}+T_{P2}^{iter}} = \frac{1}{2t+x}$. Since we have 2 parallel streams, the throughput is $\tau'_{out} = \frac{2}{2t+x}$. If we want to know when splitting and merging is worse compared to the initial PPN, then we have: $\frac{2}{2t+x} < \frac{1}{t+x}$. From this inequality it follows that $x < 0$, which contradicts with the assumption that the network is imbalanced, i.e., $x > 0$. Thus, the new throughput is the same or better than the initial PPN, i.e., $\tau'_{out} \geq \tau_{out}$.

We have shown that $\tau'_{out} = \tau_{out}$ when the initial network is already balanced and $\tau'_{out} \geq \tau_{out}$ when this is not the case. In other words, applying our approach results in performance gains when there is something to be gained by load balancing. Next, we discuss how our approach exploits data-level parallelism for PPNs with cycles and/or stateful processes.

5.3.1 Stateful Processes

When a stateful process is split up, then the different process partitions must communicate data as a result of a dependency between different process iterations. The question whether partitions of a split up process have overlapping executions or not depends on the *distance*, in terms of a number of process firings, between data production and consumption. If data is produced by a process for the next firing of the same process (i.e., the distance is 1), then there is no data-level parallelism to be exploited and splitting such a process results in sequential execution of the process partitions. However, when the distance is larger than 1, then some copies of that process have some data parallelism that can be exploited by the process splitting transformation. If, for example, the distance between data production and consumption is 5, then 5 process firings can be done in parallel before communication and synchronization is required again. Applying our solution approach, splits up all processes first. As a result, the same functions are executed by several process partitions. The necessary FIFO communication channels are automatically derived in case the split up processes are stateful. In this way, the different process partitions overlap their firings when this is allowed by the self-dependences, i.e., the dependence distance is larger than 1, and synchronize their firings when necessary.

5.3.2 Cycles

For transforming processes that form a topological cycle, it is important to realize that the process splitting and merging transformations do not re-time any of the process firings. This means that the process firings are not re-scheduled, but only assigned to different process partitions. Therefore, a cycle present in the initial PPN, will *not* be removed by our approach and the transformed PPN will have a cycle as well. The behavior of the cycle is the most important factor that determines whether performance improvements are possible or not and we illustrate this with 3 different examples in Figure 5.10. There are 2 extremes: the first is a true cycle for which nothing can be gained, and the second is a doubling of the throughput by creating 2 independent streams. A third example shows a network that gives performance results between

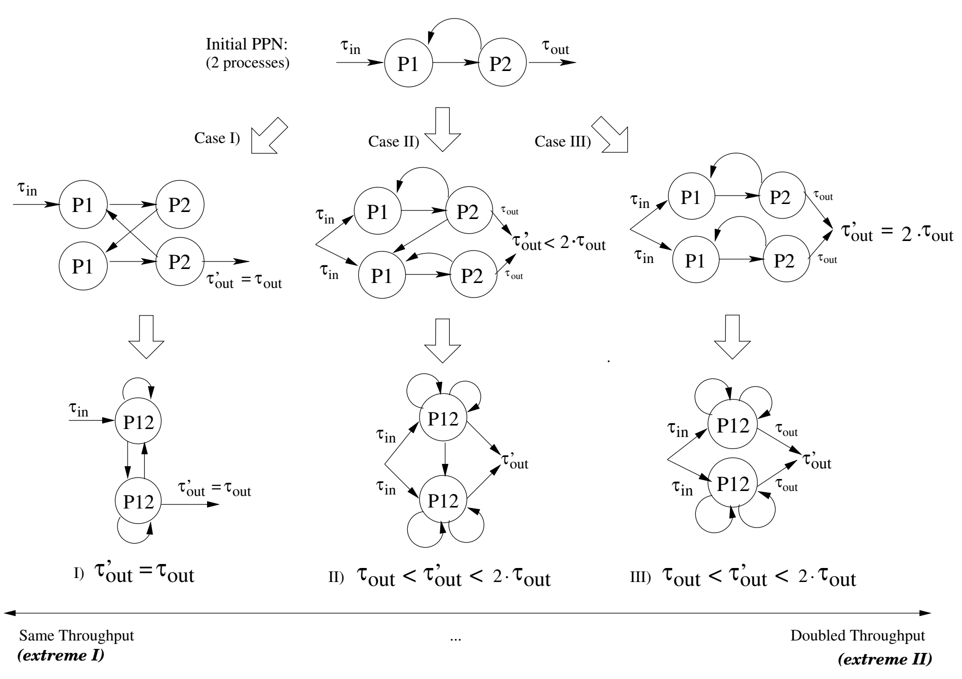

Initial PPN:
(2 processes)
Case I)
Case II)
Case III)
I) $\tau'_{out} = \tau_{out}$
II) $\tau_{out} < \tau'_{out} < 2\cdot\tau_{out}$
III) $\tau_{out} < \tau'_{out} < 2\cdot\tau_{out}$
Same Throughput
(extreme I)
Doubled Throughput
(extreme II)

Figure 5.10: Throughput Possibilities after Splitting a Cycle 2 Times

the two extremes. For the three examples in Figure 5.10, we discuss how: *i)* the initial load balancing, and *ii)* the inter-process dependencies after splitting play a role on the performance results.

Extreme I (same throughput): We already mentioned that for true cycles all processes involved in such a cycle execute sequentially. That is, data is typically read once from outside the cycle and then data is produced/consumed for/from processes belonging to that cycle. For the initial PPN in Figure 5.10, this can mean that *P1* reads from its input channel once, and then produces/consumes from the 2 channels to/from *P2*. If *P1* injects a data token in the cycle in one firing and reads a token from the feedback channel in the next firing, then processes *P1* and *P2* execute in a pure sequential way. It is clear that for this type of cycles, performance gains are not possible. Applying our solution approach on a true cycle, as shown with `Case I` in Figure 5.10, gives the same performance results as the initial PPN. The reason is that after splitting, the cycle is present as a path connecting *P1*, *P2*, *P1*, *P2*, *P1*, and after merging this sequential firing sequence is not changed as the dependencies and sequential execution do not allow any overlapping executions.

Extreme II (doubled throughput): Another extreme is a transformed network with independent data paths. The initial PPN from which this transformed PPN is derived, is topologically the same as the initial PPN in `Case I`, but the behavior is different, i.e., it is not a true cycle because *P1* injects first, for example, at least 2 tokens before reading data from the cycle. Thus, depending on the behavior of the cycle, splitting processes can result in different paths where the cycle connects only processes in the same path. In other words, independent streams can be created as illustrated with `Case III` in Figure 5.10. This can easily happen when we split processes, for example, 2 times such that the even executions of that process are assigned to one process partition, and the odd executions to another partition. If the cycle and thus the dependent producer and consumer executions are from even to even executions and from odd to odd executions, then the communication remains local to one data path as shown in `Case III` of Figure 5.10. This is an example of a cyclic PPN that has the potential to scale linearly with the number of created streams. Having a transformed PPN with independent data paths, however, does not automatically mean that performance gains are possible. Besides the dependencies as we have just discussed, the workload balancing of the initial PPN is another important factor. For our example with the 2 independent data paths, it can still happen that the same throughput as the initial network is achieved, i.e., $\tau'_{out} = \tau_{out}$, when the initial network is already perfectly balanced. That is, for a network that is already balanced, there is nothing to be gained with load-balancing. On the other hand, when the two processes are highly imbalanced, then a doubling of the throughput can be approached.

Between the 2 Extremes: The last case to be discussed from Figure 5.10, is `Case II` that gives performance results between the two extremes as discussed above. After splitting and merging, the compound processes are connected with one communication channel. Depending on how many times synchronization and data communication occurs between the compound processes, the performance results can be the same as for a true cycle (i.e., sequential execution), or the performance results can approach a doubling of the throughput if synchronization does not play a role as, for example, data is communicated only once.

5.4 Case-Studies

To illustrate that our approach works for PPNs with stateful processes and cycles, we consider 2 different algorithms and implement their initial PPN and transformed PPNs onto the ESPAM platform prototyped on a Xilinx FPGA [60], [61]. We measure the performance results to check that indeed the maximum performance gains are obtained allowed by inter-process dependencies. First, we focus on the QR algorithm, which is a matrix decomposition algorithm that is interesting as the compute processes have self-edges (stateful processes) and, in addition to this, the PPN is cyclic. Second, we consider a simple pipeline of processes and we show that our approach is as good as the initial network if the network is already perfectly balanced.

5.4.1 QR Decomposition: a PPN with Stateful Processes and Cycles

A QR decomposition of a square matrix A is a decomposition of A as $A = QR$, where Q is an orthogonal matrix and R is an upper triangular matrix. Our implementation and corresponding PPN is shown in Figure 5.11 A). It consists of 2 source processes, 1 sink process, and 2 compute processes denoted by V and R. This network is highly imbalanced as process R fires more times and is also computationally more intensive than V. Applying the process splitting transformations on processes V and R gives as a result the network shown in Figure 5.11 B). We apply our solution approach and merge process partitions of V with R (and not V with V) to create compound processes $VR1$ and $VR2$. We do this by considering first one partition of V in the network and see that it has outgoing FIFO channels to another partition of V and to one partition of R. These two process partitions are merged and in a similar way the second compound process is created. The final result and transformed PPN is shown in Figure 5.11 C). In all our experiments, we assume that source and sink processes cannot be transformed. The reason is that, for example, these processes read and write data from/to a memory location, which can only be done by one process sequentially and, thus, not by multiple processes in parallel.

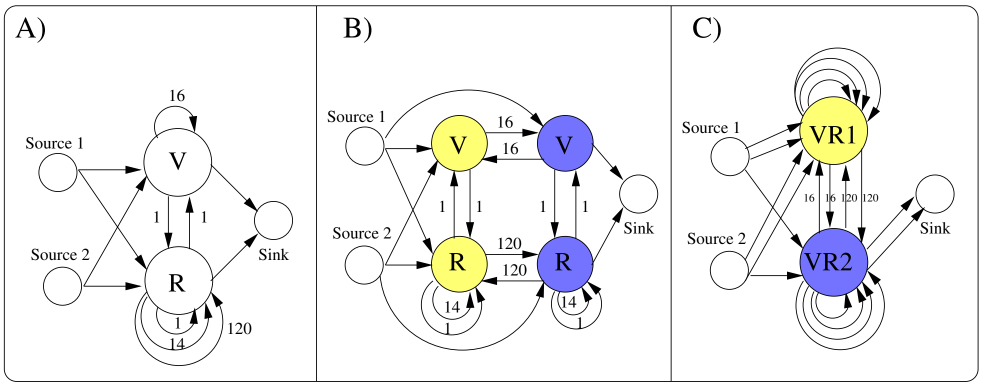


Figure 5.11: A) Intial PPN for QR Decomposition Algorithm, B) PPN with split up processes V and R, and C) load-balanced PPN with compound processes.

The resulting network is perfectly balanced. To implement the network, we apply a one-to-one mapping of processes to processors and thus 5 processors are used in total. To be more specific, the processes are executed as software routines on soft-core MicroBlaze processors, which are point-to-point connected. Figure 5.12 shows the corresponding measured performance results on the ESPAM platform [60], [61], prototyped on a Xilinx FPGA. The source and sink processes both finish one process iteration in only 1 time unit, while the compute processes V and R are the computationally intensive processes which need respectively 100 and 450 time units for one process iteration.

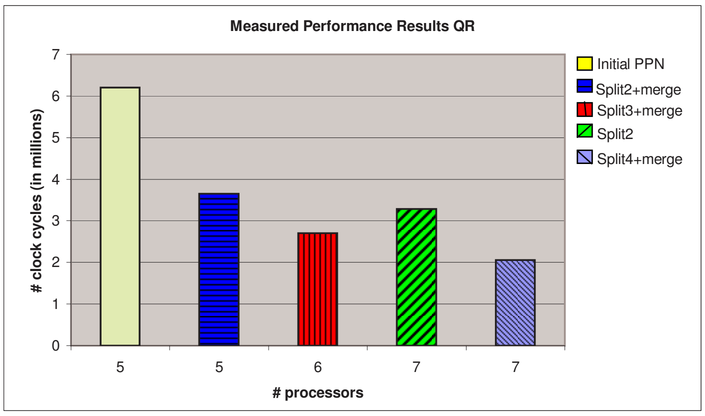


Figure 5.12: Measured Performance Results of QR on the ESPAM Platform

The first bar serves as our reference point and it corresponds to the performance results of the initial PPN shown in Figure 5.11 A). The QR network needs around 6 million cycles to finish its execution and uses 5 processors. For the same number of processors, our transformation approach gives much better performance results as shown by the second bar; the compute processes are split-up 2 times and different partitions are merged, which is denoted by *split2+merge* and shown in Figure 5.11 C). When we apply our approach and create 3 compound processes, denoted by *split3+merge*, then we even further improve performance results using 6 processors as shown by the third bar. Next, we compare the results of applying only the process splitting transformation, denoted by *split2* and shown in Figure 5.11 B), with our approach of splitting processes 4 times and merging different process partitions into compound processes, denoted by *split4+merge*. Both experiments use 7 processors and the 4th and 5th bars show the corresponding performance results. It can be seen that creating balanced partitions gives better performance results than applying only the splitting transformation. Note that the initial PPN with 5 processors executes mostly in a sequential way, i.e., no data-level parallelism is exploited. By applying our approach, i.e., splitting the compute processes 2, 3, and 4 times, we exploit data level parallelism and achieve speed ups of 1.7, 2.3, and 3, respectively.

The QR algorithm is an example of `Case II` in Figure 5.10. The self-edges in Figure 5.11 A) are annotated with their minimum buffer size capacity as computed by the `pn` compiler [95]. Process V, for example, has a self-channel that should have a capacity of at least 16 tokens to avoid a deadlock. This means that 16 tokens are produced and buffered before they are finally consumed by the same process: 16 firings of that processes could be done in parallel before data communication and synchronization are required again. We showed results for splitting up the stateful processes 2, 3, and 4 times in the experiments. After applying our approach, we see in Figure 5.11 C) that the self-channels appear as the channels connecting the compound processes. These observations make clear that the cycles are not true cycles as we have discussed in the previous section and that there is data-level parallelism to be exploited by applying our solution approach. This is, indeed, confirmed by the measured performance results. Our approach almost scales linearly by increasing the number of compound processes (2nd, 3rd, and 5th bars in Figure 5.12) compared to the initial PPN, indicating that we exploit all available data-level parallelism.

5.4.2 Transforming Perfectly Balanced PPNs

We have shown that stateful processes and cycles in PPNs restrict data-level parallelism and thus influence performance results. In this section we show that the process workload, and thus the process iteration time $T_{P_i}^{iter}$, is another aspect that should be taken into account. To illustrate this, we consider a simple PPN consisting of a pipeline of 4 processes. The goal of this experiment is to verify that our approach, compared to applying only the process splitting transformation, does not give worse performance results for PPNs that are already balanced. To check this, we generate the following 4 PPNs as also shown in Figure 5.13: *i)* the initial PPN, *ii)* a PPN with process $P2$ split up 2 times, *iii)* a PPN with processes $P2$ and $P3$ split up 2 times and different partitions merged, and *iv)*, a PPN with processes $P2$ and $P3$ split up 3 times and different partitions merged.

For each process network, we vary the workload of process $P3$ and assign 4 different values. As a result, the process iteration time T_{P3}^{iter} is 1, 50, 75, and 100 time units. This means that process $P2$ is the bottleneck when T_{P3}^{iter} is 1, 50, and 75 time units. By increasing it to 100, both $P2$ and $P3$ are equally computationally intensive. Recall that we do not transform source and sink processes $P1$ and $P4$ in our experiments. We therefore say that the network is *imbalanced* when T_{P3}^{iter} is 1, 50, or 75 time units, and *balanced* when we choose T_{P3}^{iter} to be 100. We expect that:

- The more balanced the network becomes by increasing the workload of $P3$, the less is gained by splitting only process $P2$ two times (network II in Figure 5.13);

- Our transformation approach (network III in Figure 5.13) gives better performance results when the network is imbalanced;

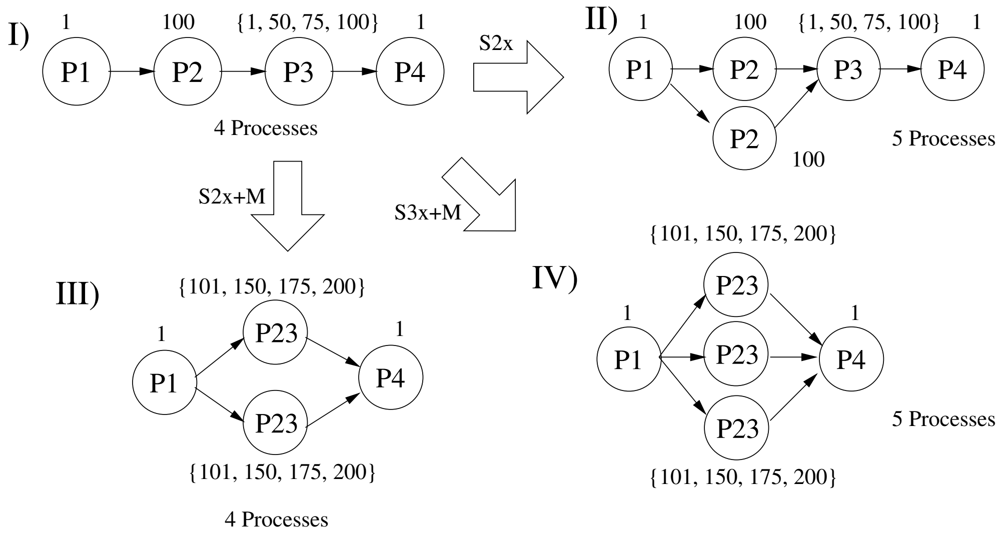


Figure 5.13: Splitting vs. "Splitting+Merging" with Different Workloads

- Our approach can even achieve better results by creating more than 2 compound processes (network IV in Figure 5.13), while this is not possible using the same number of processors and thereby applying only the process splitting transformation.

We make 2 comparisons and measure the performance results on the ESPAM platform of PPNs with an equal number of processes, i.e., PPNs with 4 processes and PPNs with 5 processes. First, we compare the initial PPN (i.e., network `I` in Figure 5.13) with the network on which process splitting and merging has been applied (i.e., network `III` in Figure 5.13). Second, we compare network `II` with network `IV` from Figure 5.13.

Figure 5.14 shows the measured performance results for the 2 different PPNs with 4 processes. The x-axis shows the different T^{iter}_{P3} configurations when the workload of process $P3$ is increased, and the y-axis the corresponding cycles counts. Because we map the processes one-to-one onto processors, there are 4 processors used in this experiment. For each workload configuration, the first bar corresponds to process network `I` in Figure 5.13 and the second bar to process network `III`. The initial PPN gives the same performance results for all different workload configurations as the overall throughput is $\tau_{out} = \frac{1}{100}$ determined by process $P2$. Our approach gives better results for unbalanced networks. However, as the workload of process $P3$ is increased, the network becomes more balanced and less can be gained by transformations targeting the same number of processors. Figure 5.14 shows that

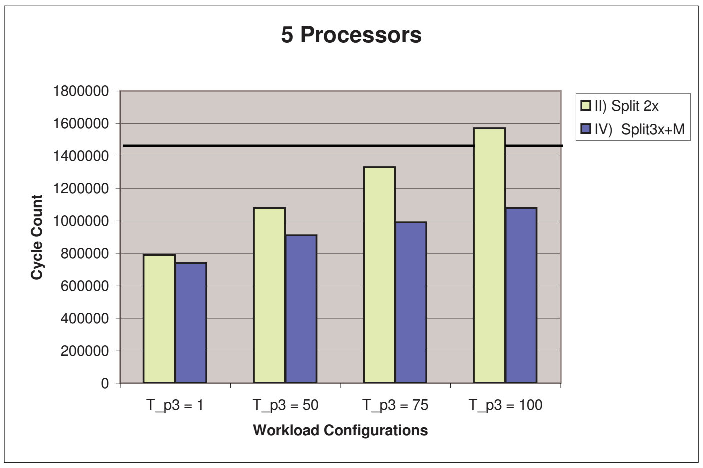


Figure 5.14: Initial PPN (PPN I) vs. Split2x + Merging (PPN III)

the difference between the initial PPN and the transformed PPN becomes smaller. The last 2 bars show the results for the PPNs where the initial network is already balanced, i.e., $T^{iter}_{P3} = 100$. It can be seen that our approach is slightly worse than the initial PPN, although the difference is not significant as it is only 2% off. The reason is that the transformations introduce a small overhead in the compound processes, which consist of additional control to execute the different functions. In the ideal case when there is no overhead, the throughput of one compound process is $\frac{1}{200}$ and thus the aggregated throughput of both compound processes is $\frac{1}{100}$, which is the same as the initial PPN. Due to the additional control, however, the process iteration time is not $T^{iter}_{P23} = 200$, but a little bit higher which finally results in the minor and not significant performance degradation. The ratio of the workload and the control overhead is important for the actual overhead and performance degradation. In our experiments, the workload of the compound processes is 200 assembly instructions. In most applications however, the process workload will be much larger such that the overhead subsequently will have less impact on the performance results and will be negligible (i.e., less than 2%).

Figure 5.15 shows the comparison between PPNs with 5 processes. That is, we compare our solution approach that splits up all processes 3 times and merges back different partitions, with applying only the process splitting transformation. For each workload configuration, the first bar corresponds to network `II` in Figure 5.13, and the second bar to network `IV`. The bold horizontal line in Figure 5.15 is the reference corresponding to the performance results of the initial PPN.

We see that applying only process splitting for process $P2$ is less beneficial as the

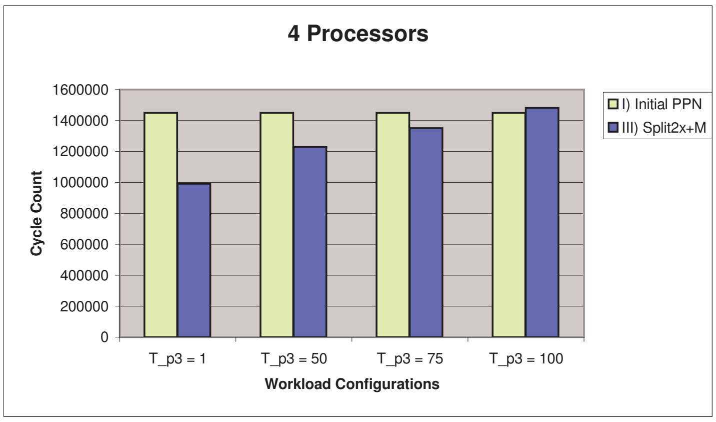


Figure 5.15: "Splitting 2x" (PPN II) vs. "Splitting 3x + Merging" (PPN IV)

network becomes more balanced as illustrated with the $1^{st}, 3^{th}, 5^{th}$, and 7^{th} bars. When the network is balanced, i.e., the 7^{th} bar, the performance results are a bit worse than the initial PPN due to some additional control introduced by the transformations as discussed before. For splitting and merging the processes 3 times, however, we see that better performance results are obtained as illustrated with the $2^{nd}, 4^{th}, 6^{th}$, and 8^{th} bars in Figure 5.15. The reason is that 3 balanced compound processes execute as 3 independent streams in parallel. Each compound process delivers tokens with a throughput of $\frac{1}{200}$ (when the time for one process iteration of processes $P2$ and $P3$ is 100 time units). The overall system throughput is therefore $\tau_{out} = \frac{3}{200} \approx \frac{1}{67}$. If only $P2$ is split up, then the overall system throughput will be determined by $P3$ and remains $\tau_{out} = \frac{1}{100}$. We see that our approach gives better performance results for all workload configurations. By increasing the workload and thus also T^{iter}_{P3}, the cycle count goes up, but not as steep compared to applying only the process splitting. In addition, our approach would also scale for more than 5 processors, as an arbitrary number of independent streams can be created.

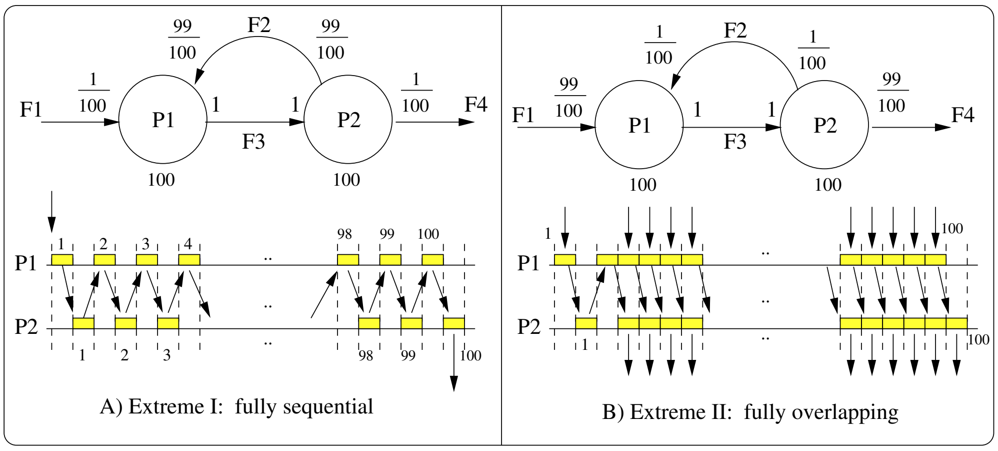
5.5 Discussion and Summary

We have shown that better performance results are obtained when both the process splitting and merging transformations are applied in combination, as opposed to applying only one of these transformations. Furthermore, we have shown that it is very difficult to identify a single bottleneck process in a PPN, since there can be many different bottleneck processes during the execution of a PPN. Our approach solves the problem of selecting a process on which the transformations have the largest impact, as first all processes are split up and then perfectly load-balanced compound processes are created using the process merging transformation. Furthermore, we have shown that our approach also works for process networks with cycles and stateful processes. If in the initial PPN there is data-level parallelism to be exploited, then our approach gives better performance results compared to the initial PPN by exploiting this parallelism to the maximum. The same performance results are obtained when no data-level parallelism is available in the initial PPN.

After applying our solution approach a designer may end up with a transformed PPN which performance is the same as the initial PPN. As already explained, the reason can be that the initial PPN is already perfectly balanced, or cycles can be present in the PPN that restrict the data-level parallelism. If we focus on cyclic PPNs, then we know that performance gains are not possible when processes involved in a true cycle are split up. This makes it clear that it is desired to indicate to the designer when a PPN contains a true cycle. Therefore, we sketch an approach how true cycles can be detected, i.e.,

- we investigate if the number of input tokens that the processes read from outside the cycle can serve as a metric to detect true cycles.

We consider the two example PPNs shown in Figure 5.16, which are different in the number of tokens read from outside the cycle.



F1
F2
F3
F4
P1
P2
99/100
1/100
100
1
A) Extreme I: fully sequential
B) Extreme II: fully overlapping

Figure 5.16: Different Behavior of a Cycle

The cyclic PPNs are topologically the same, but the behavior of the cycles are different. That is, processes *P1* and *P2* both have 100 process iterations, but the cyclic PPNs are different in the total number of input tokens read from processes that are involved in the cycle. In Figure 5.16 A), process *P1* reads data only once from a process that is not part of the cycle (i.e., the process writing to FIFO channel $F1$), and 99 times from FIFO channel $F2$ that is written by $P2$, i.e., a process involved in the cycle. These channels are therefore, respectively, annotated with the fractions $\frac{1}{100}$ and $\frac{99}{100}$. The behavior of process $P1$ is the following: it injects one token in the cycle in one iteration, and in a next iteration it needs to read a token from the cycle first, before it can inject one token again. This leads to sequential execution of both processes, as illustrated with the time lines of $P1$ and $P2$ in Figure 5.16 A). From this example, we learn that a cycle is a true cycle when the processes read the input data only *once* from outside the cycle and then *always* read/write from/to the cycle.

The other extreme is shown in Figure 5.16 B), where all the input data is read from outside the cycle, except only one input token. Thus, topologically the PPN in Figure 5.16 B) is the same as in A), but the behavior of the cycle is different. That is, process $P1$ reads 99 tokens from FIFO $F1$ that is *not* part of the cycle, and only once from FIFO $F2$ that is part of the cycle. This makes both processes $P1$ and $P2$ from that point of view independent, i.e, the cycle does not sequentialize the executions of $P1$ and $P2$, which results in overlapping execution of both processes as illustrated with the time lines in Figure 5.16 B). This example shows that the cycle is not a true cycle, because all the input data (except one token) is read from outside the cycle.

From the two extreme cases presented in Figure 5.16, we learn that the number of input tokens that the processes read from outside the cycle, can serve as a metric to detect the behavior of a cycle, i.e., whether it is a true cycle. If only one token is read from outside and all the others are read/written from/to the cycle, then the cycle is a true cycle. A true cycle should be easy to detect at compile-time with similar techniques presented in the previous chapters, i.e., polyhedral analysis and counting integer points in polyhedral descriptions of input/output port domains. Thus, a designer can be informed when a cyclic PPN contains a true cycle for which performance gains are not possible. Besides the information on the behavior of the cycles, a designer may also be interested in *how much parallelism* there is, in case there is something to be gained. Therefore, we investigate whether the number of input tokens that are read from outside a cycle, is also an indication how much the processes inside the cycle can overlap their executions.

Let us consider an example where a process reads *half* of its input tokens from outside the cycle, and the other half from inside the cycle. A cyclic PPN with this behavior is shown in Figure 5.17 A). Similar to the example in Figure 5.16, the processes have 100 iterations, but now process $P1$ reads in total 50 tokens from input port $IP1$, i.e., from outside the cycle, and it reads in total 50 tokens from input port $IP2$, i.e., from a process that is part of the cycle. The FIFO channels are therefore annotated with the fractions $\frac{50}{100}$. With this example, we want to indicate that *i)* the communication pattern's influence on the behavior of cycles, and *ii)* that these communication patterns are important for all processes involved in the cycle.

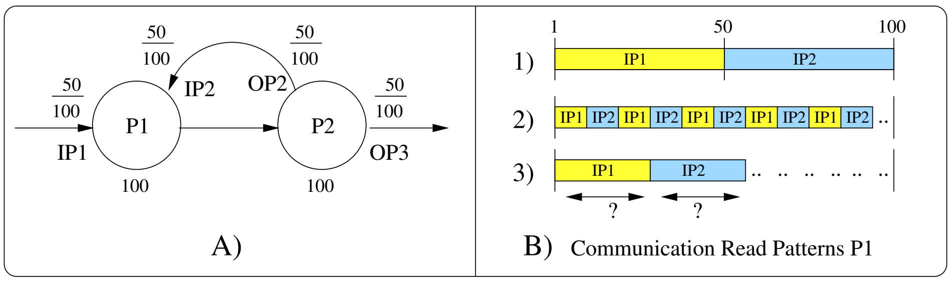

50
100
50
100
50
100
50
100
IP1
P1
IP2
OP2
P2
OP3
100
100
A)
1
50
100
1)
IP1
IP2
2)
IP1 IP2 IP1 IP2 IP1 IP2 IP1 IP2 IP1 IP2 ..
3)
IP1
IP2
?
?
B) Communication Read Patterns P1

Figure 5.17: What is the behavior when half of the data is read from outside?

Three different communication patterns are shown in Figure 5.17 B), which classifies how process *P1* can read its input data from two input ports *IP1* and *IP2*. Example `1)` shows the time line of process *P1* that has 100 process iterations. Process *P1* needs one input token from either one of its two input ports per process iteration. It reads consecutively 50 tokens from input port *IP1* in the first 50 iterations. Then, 50 tokens from *IP2* are consecutively read in the remaining 50 process iterations. A different pattern is shown with example `2)`. In one process iteration, one token is read from input port *IP1*, and in the next iteration one token is read from *IP2*, which is repeated 50 times. Thus, the tokens are read one by one from different input ports. Example `3)` does not read all tokens consecutively from one port as in example `1)`, it also does not read only 1 token as in example `2)`, but a number of tokens between these extremes.

Figure 5.18 shows the overlap of the two processes involved in the cycle that read/write data with the different communication patterns as we have identified above, i.e., it shows the time lines of processes *P1* and *P2*. We experiment with different communication patterns selected from Figure 5.17 B) and want to show that there is overlap to some extent in all the examples. Each block in the time lines corresponds to one process iteration, i.e., the yellow, blue, white, and red boxes. The executions of *P1* are annotated with the input ports from where *P1* reads its input data (i.e., *IP1* or *IP2*). And the executions of *P2* are annotated with the output ports where *P2* writes its output data to (i.e., output port *OP2* or *OP3*). The arrows denote dependencies, i.e., how data is communicated, and thus a simulation of the cyclic PPN is shown.

There are many combinations of different communication patterns possible for the processes involved in the cycle, because a process does not only have 3 options to read/write in a particular pattern, but these patterns can also be ordered differently inside each process (see process *P2* in Figure 5.18 A and B). Figure 5.18 shows some representative examples of a cycle with different communication patterns and it also illustrates that the overlap in process executions for some examples is minimal (e.g., in Figure 5.18 C), while the overlap for some other examples is substantial (e.g.,

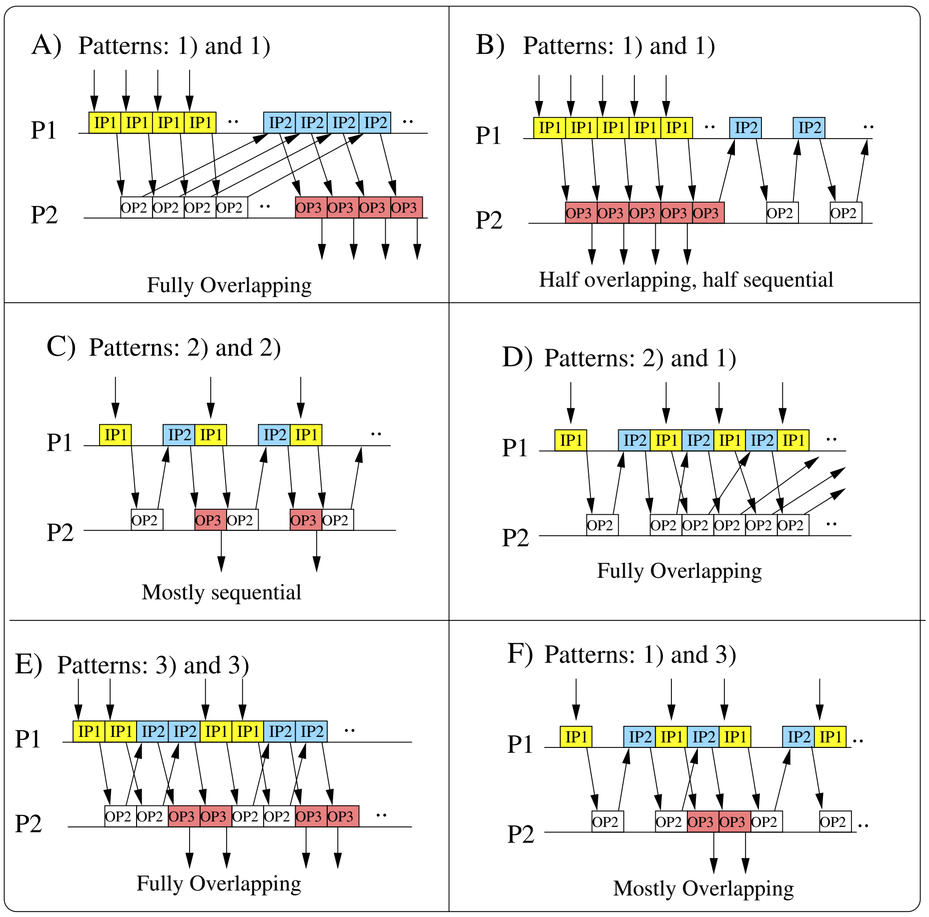


Figure 5.18: Behavior of a Cycle with Different Read/Write Patterns

in Figure 5.18 A), D) and E)). Note that the number of input data that is read from outside the cycle is the same for all examples, i.e., 50 tokens, while the behavior of the cycles are very different. Therefore, we conclude that the number of input tokens read from outside the cycle cannot serve as a metric how much the processes overlap and thus how much can be gained with applying our solution approach. More sophisticated analysis is required, which is therefore left for future research.

Recall that our approach first splits up all processes in a PPN before process instances are merged back (see Algorithm 2). Our final remark is about the order in which the process splitting transformation is applied consecutively on all processes. That is, we did not investigate whether applying the splitting transformation in a different order has an effect on, for example, the number of FIFO channels and/or the final performance results of the transformed PPN. This is left for future research.