Combining Process Splitting and Merging Transformations for Polyhedral Process Networks

Sjoerd Meijer Hristo Nikolov Todor Stefanov
LIACS, Leiden University, The Netherlands
{smeijer,nikolov,stefanov}@liacs.nl

ABSTRACT

We use the polyhedral process network (PPN) model of computation to program and map streaming media applications onto embedded Multi-Processor Systems on Chip (MPSoCs) platforms. In previous works, it has been shown how to apply different process network transformations in isolation. In this work, we present a holistic approach combining the process splitting and merging transformations and show that it is necessary to use both transformations in combination to achieve the best performance results, which cannot be achieved using only one transformation. We solve the problem of ordering both transformation and, in addition, relieve the designer from the task to select the processes on which the transformation should be applied. Thus, our approach combines both transformations exploiting the data-level parallelism available in a PPN as much as possible, even in cases where the parallelism is restricted by topological cycles and stateful processes in the PPN.

I. INTRODUCTION

The programming of streaming media applications for embedded Multi-Processor System on Chips (MPSoCs) is a notorious difficult and time consuming task as it involves the partitioning of applications and synchronization of different program partitions. To address these issues, the `pn` compiler [1] has been developed. It derives Polyhedral Process Networks (PPNs) from sequential nested-loop programs. This is illustrated in Figure 1 (denoted by arrow `I`). If, for example, the input is a sequential program with 3 program statements, then the output of the `pn` compiler is a PPN consisting of 3 processes.

Polyhedral Process Networks [2] is a special class of Kahn Process Networks (KPNs) [3]. A PPN consists of autonomous processes that communicate and synchronize over FIFO channels using blocking FIFO read and write primitives. The functional behavior of each process is expressed in terms of polyhedral descriptions. Thus, everything about the execution is known at compile-time, which allows the calculation of buffer sizes and schedules for merging processes. Applications specified as (polyhedral) process networks allow a more natural mapping of processes to processing elements of the MPSoC architecture than a sequential program specification [4]. In the `pn` partitioning strategy, a process is created for each function call statement in the nested loop program as shown in the example in Figure 1. Such a partitioning strategy may not necessarily result in a PPN that meets the performance or resource requirements. To meet these requirements, a designer

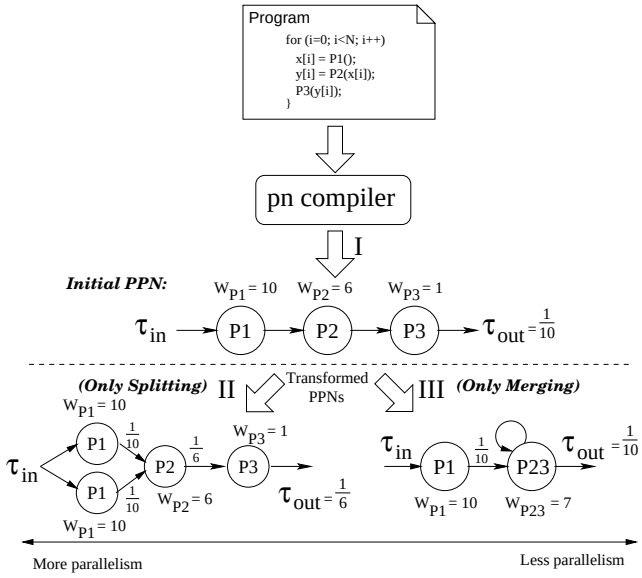

Fig. 1. Deriving and Transforming Process Networks

can apply transformations to increase parallelism by splitting processes as defined in [5], or to decrease parallelism by merging processes into a single component as defined in [6]. In Figure 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 $P2$. Arrow `III` is an example of transforming the initial PPN by applying the merging transformation on processes $P2$ and $P3$ to create compound process $P23$. Although the process splitting and merging transformations have been defined in [5], [6], no hints were given how to apply these transformations. This is necessary as there are many options to apply a transformation and many factors should be taken into account to achieve good performance results. The problem how to apply each transformation has been addressed in [7] and [8], where compile-time approaches for each of these transformations have been defined in isolation. 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.

Problem Definition and Paper Contributions

The pn compiler derives Polyhedral Process Networks (PPNs) from sequential program specifications. Recall that there are two parameterized transformations that play a vital role in meeting the performance/resource constraints: *i)* the *process splitting* transformation to create multiple instances of the same process to better distribute the workload, and *ii)* the *process merging* transformation to reduce the number of processes in the network by sequentializing them in a compound process. The former 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 latter, 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 III-B. 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 paper we:

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

II. Motivating Examples

In this section, we investigate whether applying both the process splitting and merging transformations in combination gives better performance results than using only one transformation. Consider the initial and transformed PPNs in Figure 1. Each process P_i is annotated with a workload number, to which we refer as W_{P_i}:

$$W_{P_i} = C^{P_i} + x \cdot C^{Rd} + y \cdot C^{Wr}, \quad (1)$$

where C^{P_i} denotes the number of time units required to execute the process function once, x and y denote how many FIFOs are read and written per process firing, and C^{Rd} and C^{Wr} denote the communication costs, i.e., the numner of time units required to read/write a single token from/to a FIFO channel. Note that the costs for the process function C^{P_i} and FIFO communication C^{Rd} and C^{Wr} are modeled with constants, because we assume that C^{P_i} is the worst case execution time of the function, and for constant communication costs we rely on MPSoC platforms that provide an interconnect with guaranteed services, i.e., the ESPAM platform [9] in our case.

Process $P1$ in Figure 1, for example, needs 10 time units, i.e., $W_{P1} = 10$, while $P2$ is computationally less intensive process as it takes 6 time units, i.e., $W_{P2} = 6$. Process $P3$ needs only 1 time unit, respectively. Process $P1$ determines therefore the system throughput of the initial PPN. The throughput is denoted by τ_{out} and we define it as the average number of tokens produced by the network per time unit. Since $P1$ is the most computationally intensive process that executes 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 1.

A. Transforming and Deriving a PPN with 4 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 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 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 faster. 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 execute its 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 $\frac{1}{6}$ ($W_{P2} = 6$). 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 execution of process $P3$. We see that $P3$ can process data much faster than it actually receives, as it only has a workload of $W_{P3} = 1$, but still it produces tokens with a throughput of $\frac{1}{6}$ caused by the slowest process $P2$. The overall system throughput is therefore $\tau_{out} = \frac{1}{6}$, determined by $P2$. Thus, we have derived a PPN which gives a throughput $\tau_{out} = \frac{1}{6}$ that is much better than the original 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 to derive the transformed PPN shown in Figure 2 A). Two independent data paths are created

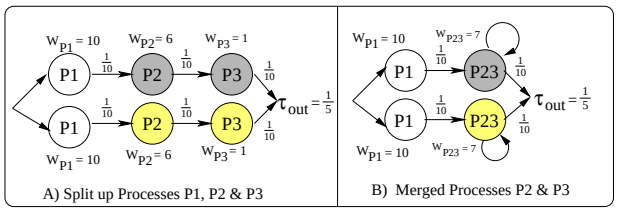

Fig. 2. Transformed PPN2: Splitting and Merging to Create 4 Processes

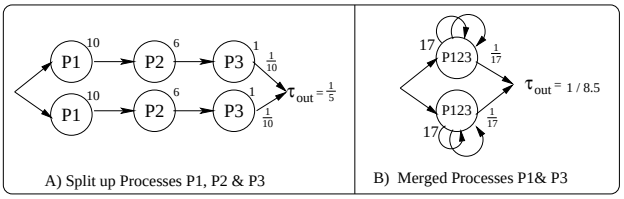

Fig. 3. Transformed PPN4: Creating 2 Load-Balanced Tasks

each consisting of 3 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 2 A) is $\tau_{out} = \frac{1}{10} + \frac{1}{10} = \frac{1}{5}$. When we merge $P2$ with $P3$, process $P1$ remains the bottleneck and the throughput is unaffected as shown in Figure 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 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 as throughput $\tau_{out} = \frac{1}{6}$. In fact, to create a PPN with n processes from the initial PPN in Figure 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.

B. Transforming and Deriving a PPN with 2 Processes

A designer sometimes needs to reduce the number of processes for a given PPN in order to meet the resource constraints. Another reason to reduce the number of 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 from Figure 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 at the bottom right in Figure 1, which is obtained by applying only the process merging transformation. The resulting network has the same throughput as the original 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 the original 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 3.

All processes are first split up twice as shown in Figure 3 A). Then, two compound processes are created by merging an instance of each process into a compound process $P123$ as shown in Figure 3 B). The workload of a compound process is $W_{P123} = W_{P1} + W_{P2} + W_{P3} = 17$ time units, because all functions are executed sequentially. This means that a compound process delivers tokens with a throughput of $\tau = \frac{1}{17}$. Since we have 2 compound processes, the resulting overall throughput is $\tau_{out} = \frac{1}{17} + \frac{1}{17} = \frac{1}{8.5}$, which is better than the throughput $\tau_{out} = \frac{1}{10}$ of our previous example *Transformed $PPN3$ (only merging)* shown in Figure 1. This is another example which shows that both transformations should be used in combination to obtain better performance results, which cannot be obtained by only one transformation, i.e., the merging transformation in this case.

C. Transformations Resulting in 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 (see Figure 1) consisting of 3 processes and split up both processes $P1$ and $P2$ to obtain the PPN shown in Figure 4 A).

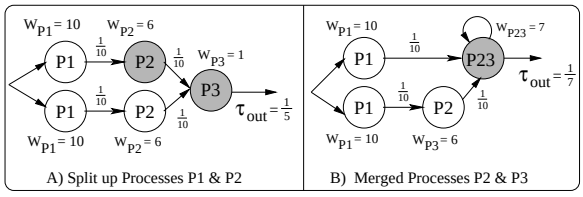

Fig. 4. Transformed PPN5: Splitting and Merging to Create 4 Processes

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 $P3$ as shown in Figure 4 B). The created compound process $P23$ has a workload of $W_{P23} = 7$ 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, it is worse than applying only the splitting transformation, i.e., *transformed $PPN1$ (only splitting)* in Figure 1 with a throughput of $\tau_{out} = \frac{1}{6}$ and subsequently also worse

than *Transformed* $PPN2$ shown in Figure 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 into a PPN with 2 processes, see *Transformed* $PPN3$ and *Transformed* $PPN4$ in Section II-B. 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 A). Then, two compound processes are created,

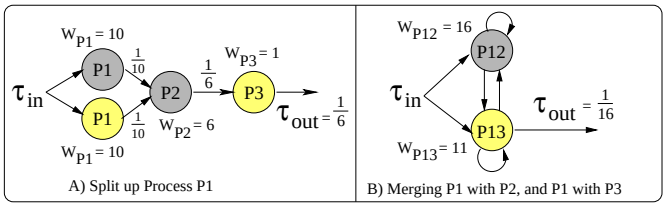

Fig. 5. Transformed PPN6: Splitting and Merging to Create 2 Processes

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 use 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, however, 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.

III. Solution Approach

Before introducing our solution in a more formal way, we show how our approach intuitively works for the examples discussed in Section II. 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 1. In two other examples, shown in Figure 2 B) and Figure 4 B), different networks were obtained by consecutively using the process splitting and merging transformations. In addition to these examples, our approach gives yet another solution as shown in Figure 6 that also gives better performance results than all 3 other PPNs mentioned above. With this example, and all the other examples presented in Section II, we have shown that we need a simple and effective solution to help the designer in transforming PPNs in an efficient way.

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,

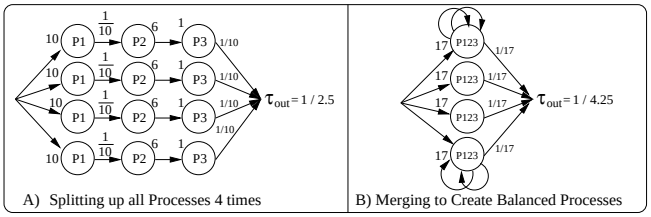

Fig. 6. Creating 4 Load-Balanced Tasks

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 6 A). In this way, we create a PPN consisting of 12 processes. Next, we merge back process instances into compound processes such that they contain one instance of each process. Figure 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 workload of the compound processes is 17 time units that is obtained by summing the workload of the individual processes (W_{P1}) and thus we see that each compound process produces $\frac{1}{17}$ tokens per time unit. Since there are 4 of such compound processes, the overall system throughput $\tau_{out} = \frac{4}{17} \approx \frac{1}{4}$, which is better than all other transformed PPNs with 4 processes shown in Figure 1, Figure 2 B) and Figure 4 B).

The initial PPN 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 3; all processes are first split up 2 times, and then compound processes are created by merging different instances such that the resulting transformed network consists of 2 processes.

A. Creating Load-Balanced Tasks

While we illustrated our solution approach with the examples in Figure 3 and Figure 6, a more formal description of this approach is given with the pseudo-code in Algorithm 1. 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 1 : 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 1 uses two functions: `split` and `merge`. For the former, we refer to [7] 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 [7] to select and perform the processes splitting. For the process merging transformation, we rely on the approach described in [8]. 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 avoids communication and synchronization of different compound processes. An example of this is given in Figure 6. One instance of $P1$ has only one channel to $P2$, which on its turn has only one channel to $P3$. Merging processes in this sequence results in compound processes that do not have any communication channels between 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 7 A), one particular consumer has to be selected and merged with the producer.

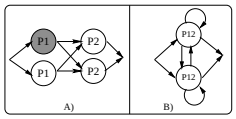

Fig. 7. Different Merging Options

If we start with the first instance of $P1$, i.e., grey process $P1$ in Figure 7 A), then we see that it has two outgoing channels to two instances of $P2$. Regardless which instance of $P2$ is chosen for merging, the resulting compound processes will have channels for data communication, as shown in Figure 7 B). In our approach, we simply consider the first outgoing channel and corresponding consumer process and merge it with the producer, because the data is evenly distributed over the channels. We mark the selected consumer as being merged already in the merging procedure, to avoid that it will be selected again.

B. Selecting Processes for Transformations

Our solution approach in Section III solves another problem indicated in Section I, i.e., how to select processes 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 most computationally intensive process. For process merging, it is important to avoid merging the bottleneck process, i.e., not introducing an even more computationally intensive process. In general, however, it is not possible to determine a single bottleneck process of a PPN. The reason is that, in PPNs, different data paths can transfer a different number of tokens. As a result, different processes can dominate the overall system throughput at different stages during the execution of the network, which we illustrate with the example shown in Figure 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

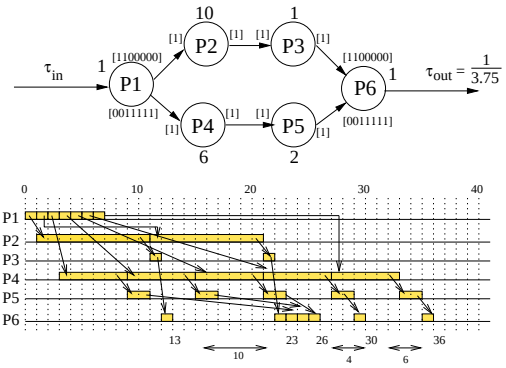

Fig. 8. What is the Bottleneck: $P2$ or $P4$?

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 executions of $P1$, but not in the remaining 5 executions. As a consequence of these patterns, more tokens are communicated through the second datapath $DP2$. At the bottom of Figure 8, the different time lines of the processes are shown. Each block corresponds to an execution 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 $P2$ largest workload of 10 time units, process $P4$ with a workload of 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 solves this problem as the transformations are applied on all processes and, therefore, it is not necessary to select particular processes.

IV. Discussion

The idea of our approach presented in Section III 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 [10], we refer to data-level parallelism when processes do not dependent on previous executions of itself. Obviously, when there is no self-edge, the process is *stateless* and an arbitrary number of independent process instances can be created that run in parallel. When a process has a self-edge, however, it produces data for itself and there exist a dependency between different executions 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.

Balanced PPNs: Let us consider a simple acyclic PPN with only two processes $P1$ and $P2$ as shown in Figure 9 A).

Case I: if a network with processes $P1$ and $P2$ is balanced, then $W_{P1} = W_{P2} = t$ time units and thus $\tau_{out} = \frac{1}{t}$. If we apply splitting and merging, as illustrated with the arrows in Figure 9 A), 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.

Case II: if a network with processes $P1$ and $P2$ is imbalanced, then we have $W_{P1} = t$ and $W_{P2} = t + x$, 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 $\tau = \frac{1}{W_{P1}+W_{P2}} = \frac{1}{2t+x}$. 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 fact that the network is imbalanced and that $x > 0$. 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 that $\tau'_{out} \geq \tau_{out}$ when this 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.

A. Stateful Processes

When a stateful process is split up, then the different process instances must communicate data as a result of dependencies between different executions. The question whether the process instances can run in parallel, i.e., they have overlapping executions, depends on the *distance* of the self-dependency, which is expressed in terms of a number of process firings between data production and consumption. If data is produced for the next execution of a 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 instances. 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 executions can be done in parallel before synchronization is required again. Applying our solution approach, splits up all processes first. As a result, the same functions are executed by several process instances. The necessary FIFO communication channels are automatically derived in case the split up processes are stateful. In this way, the different process instances overlap their executions when this is allowed by the self-dependences, i.e., the dependence distance is larger than 1, and synchronize their executions when necessary.

B. Cycles

For splitting 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 executions. This means that the process executions are not re-scheduled, but only assigned to different process instances. 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 9. 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 the two extremes. For the three examples in Figure 9, 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 9, 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 execution and reads a token from the feedback channel in the subsequent executions, 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 `Example I` in Figure 9, 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 execution sequence is not changed as the dependencies and sequential execution does not allow any overlapping executions.

Extreme II (doubling 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 *Extreme 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

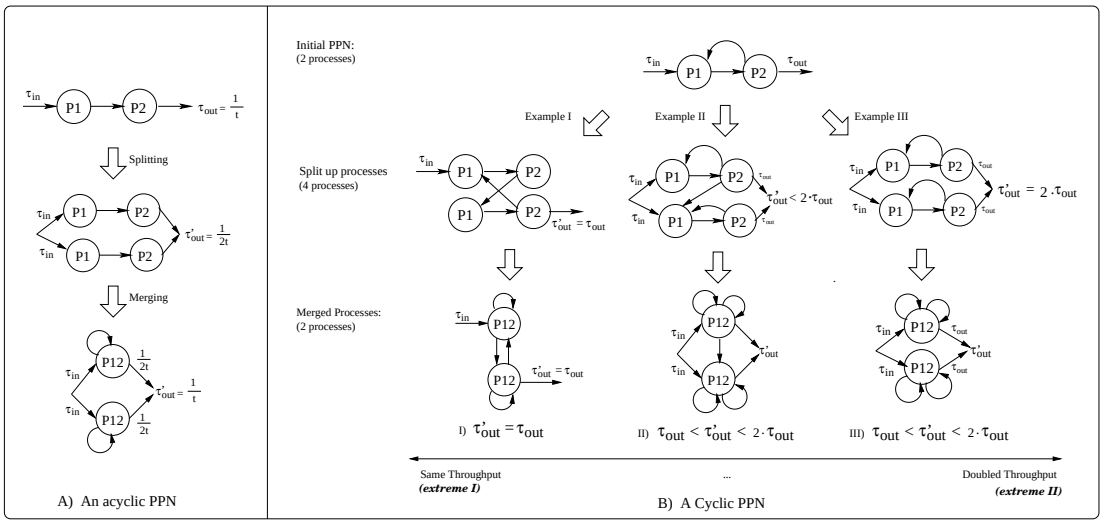

Fig. 9. Throughput Possibilities after Splitting 2 Times

the cycle connects only processes in the same path. In other words, independent streams can be created as illustrated with `Example III` in Figure 9. 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 instance, and the odd executions to another instance. 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 `Example III` of Figure 9. 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: `Example II` in Figure 9 gives performance results between the two extremes. 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.

V. Results

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 [11], [9]. We measure the performance results to check that indeed the maximum performance gains are obtained when 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 and, in addition to this, the PPN is cyclic. Secondly, 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.

A. QR Decomposition - a PPN with Process State 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 10 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 executes 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 10 B). We apply our solution approach and merge instances of V with R (and not V with V) to create compound processes $VR1$ and $VR2$. We do this by considering first one instance of V in the network and see that it has outgoing FIFO channels to another instance of V and to one instance of R. The remaining two process instances of V and R are merged and in a similar way to create the second compound process. The final result and transformed PPN is shown in Figure 10 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.

The resulting network is perfectly balanced. To implement

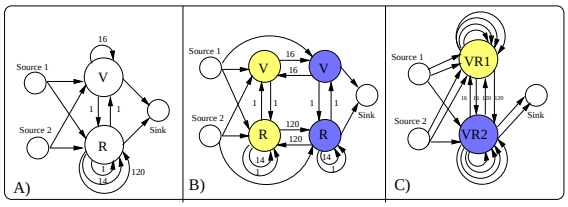

Fig. 10. A) Intial PPN for QR, B) PPN with split up processes V and R, and C) load-balanced PPN with compound processes.

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 mapped as software threads onto softcore MicroBlaze processors, which are point-to-point connected. Figure 11 shows the corresponding measured performance results on the ESPAM platform [11], [9], prototyped on a Xilinx FPGA. The source and sink processes have a workload of only 1 instruction, while the compute processes V and R have workloads of 100 and 450 instructions, respectively.

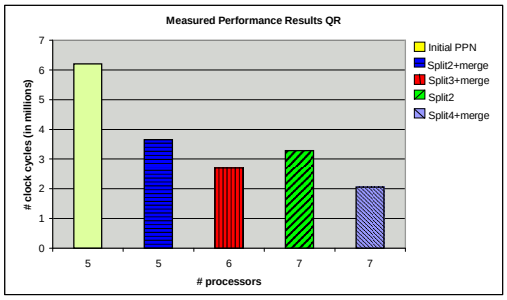

Fig. 11. Measured Performance Results of QR on the ESPAM Platform

The first bar serves as our reference point and it corresponds to the performance results of the initial PPN shown in Figure 10 A). The QR network needs 6 million cycles to finish its execution and uses 5 processors. For the same number of processors, our approach is much better as shown by the second bar; the compute processes are split up 2 times and different instances are merged, which is denoted by *split2+merge* and shown in Figure 10 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 10 B), with our approach of splitting processes 4 times and merging different instances into compound processes, denoted by *split4+merge*. Both experiments use 7 processors and the 4^{th} and 5^{th} 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 `Example II` in Figure 9. The self-edges in Figure 10 A) are annotated with their minimum buffer size capacity as computed by the `pn` compiler [1]. 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 executions of that process could be done in parallel before synchronization is required again, while 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 10 C) that the self-channels appear as the channels connecting the compound processes. These observations makes 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 and 5th bars) compared to the initial PPN, indicating that we exploit all available data-level parallelism.

B. 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 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 12: *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 instances merged, and *IV)*, a PPN with processes $P2$ and $P3$ split up 3 times and different instances merged.

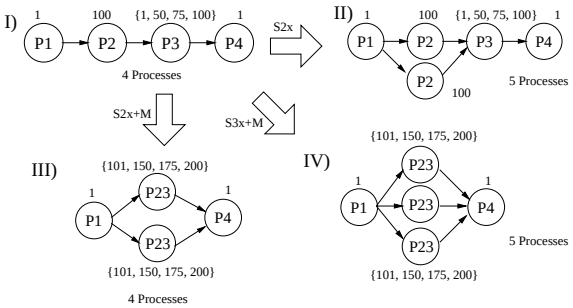

Fig. 12. Splitting vs. "Splitting+Merging" with Different Workloads

For each process network, we vary the workload of process $P3$ and assign 4 different values: 1, 50, 75, and 100 time units. This means that process $P2$ is the bottleneck when $P3$ has a workload of 1, 50, and 75 time units. By increasing it to 100, eventually 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 $P3$ has a workload of 1, 50, or 75 time units, and *balanced* when we choose the

workload to be 100 for $P3$. We expect that: *i)* 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), *ii)* our approach (network III) provides the same performance as the initial PPN when the network is already balanced, *iii)* our approach can even achieve better results by creating more than 2 compound processes (network IV), 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 12) with the network on which process splitting and merging has been applied (i.e., network `III` in Figure 12). Second, we compare network `II` with network `IV` from Figure 12.

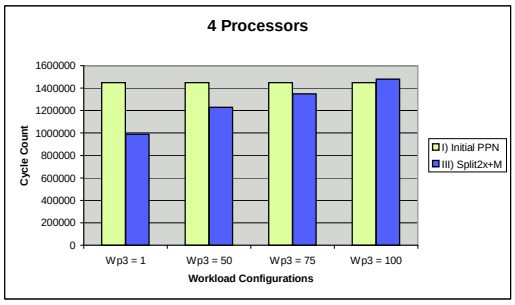

Fig. 13. Initial PPN (PPN I) vs. Splitting 2x + Merging (PPN III)

Figure 13 shows the measured performance results for the 2 different PPNs with 4 processes. 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 12 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 13 shows that the difference between the initial PPN and the transformed PPN becomes smaller. The last 2 bars displays the results for the PPNs where the initial network is already balanced, i.e., workload configuration $W_{P3} = 100$. 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 workload is not $W_{P23} = 200$, 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 will, therefore, have less impact on the performance results and will be negligible (i.e., less than 2%).

Figure 14 shows the comparison between PPNs with 5 processes. That is, we compare our solution approach that splits up all processes 3 times and merge back different instances, with applying only the process splitting transformation. For each workload configuration, the first bar corresponds to network `II` in Figure 12, and the second bar to network `IV`. The bold horizontal line in Figure 14 is the reference corresponding to the performance results of the initial PPN.

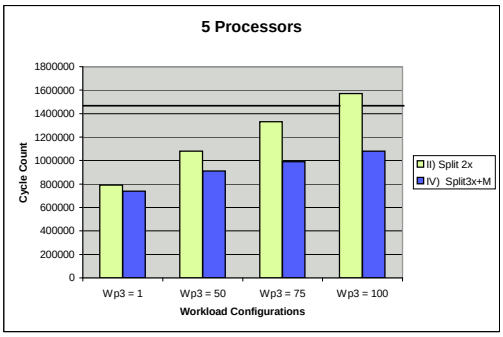

Fig. 14. "Splitting 2x" (PPN II) vs. "Splitting 3x + Merging" (PPN IV)

We see that applying only process splitting for process $P2$ is less beneficial as the network becomes more balanced as illustrated with the $1^{st}, 3^{th}, 5^{th}$, and 7^{th} bars. Eventually, 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 14. 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 processes $P2$ and $P3$ have a workload of 100). 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, 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.

VI. Related Work

Our work is most closely related to the work of [10] that aims at exploiting coarse-grained task, data and pipeline parallelism in stream programs. The StreamIt [12] compiler derives

stream graphs which are mapped on the Raw architecture and has optimizations for filter fusion and fission [13], comparable to our process merging and splitting transformations. In their approach, fusion is performed as long as the result of each fusion is stateless. When the filter becomes stateful, fission is performed on the stateless and coarsed-grain task to create more data-level parallelism. We have shown in this paper, however, an approach for cycles and networks with stateful processes and demonstrated that performance gains are possible. Two other approaches that unfold and partition processes in streaming applications using integer linear programming techniques are described in [14], [15]. However, they again work only on acyclic graphs and stateless processes.

Another difference is that we derive process networks from sequential programs written in C, as opposed to the StreamIt language that has language constructs to specify filters and FIFO communication. Each kernel in the StreamIt language has a single input and single output channel. Our process networks, however, can have multiple input/output channels and can read/write all, or a subset, of these channels in a single execution of a process.

In [16], another approach is shown for mapping stream programs onto heterogeneous multiprocessor systems. A partitioning algorithm is presented that takes as input a graph, and outputs a mapping to fuse kernels to tasks. In an iterative manner, tasks are merged, kernels are moved from bottleneck processors, and tasks are created. Similar to the StreamIt approach, an annotated version of the C programming language is used. Moreover, only stateless kernels are split for greater parallelism. Besides the average load of each kernel on each processor, similar to the workload of our processes, an additional parameter is required obtained from run-time analysis which we do not need. That is, the average data rate on each stream that must be obtained from a profile.

In [17], the scheduling of Synchronous DataFlow (SDF) graphs [18] to parallel targets is discussed. The work focuses on partitioning and scheduling techniques that exploit task and pipeline parallelism. To schedule a SDF graph, a precedence graph is first constructed, which exposes the available data level parallelism. Then, to limit the explosion of nodes, clustering is applied and thus composite nodes are created. A fundamental difference with our work is that workloads are not taken into account in the clustering as we presented in this paper. In addition to this, polyhedral process networks are more expressive than SDFs. In SDFs, all incoming and outgoing channels are read/written per firing of an actor, while in PPNs FIFO reads/writes can occur in some patterns and are described by (parameterized) polytopes. The fact that FIFOs can be read/written in some patterns, is similar to the cyclo-static dataflow graphs (CSDF) [19], with the difference that the phases in PPNs can be very large, as they are derived from nested-loop programs, and can also be parametric.

VII. Conclusion

In this paper 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 PPNs, since this can vary during the execution of a PPN. Our approach solves this problem, as first all processes are split up and then perfectly load-balanced compound processes are created using the process merging transformation. 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, and the same performance results when no data-level parallelism is available.

Acknowledgements

The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement no. 100029 and from SenterNovem.

References

[1] S. Verdoolaege, H. Nikolov, and T. Stefanov, "pn: a tool for improved derivation of process networks," *EURASIP J. Embedded Syst.*, no. 1, pp. 19–19, 2007.

[2] "To appear in handbook of signal processing systems, download via: https://lirias.kuleuven.be/handle/123456789/235370."

[3] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," in *Proc. of the IFIP Congress 74*. North-Holland Publishing Co., 1974.

[4] E. A. de Kock, "Multiprocessor mapping of process networks: a jpeg decoding case study," in *Proc. of ISSS*, 2002, pp. 68–73.

[5] T. Stefanov, B. Kienhuis, and E. Deprettere, "Algorithmic transformation techniques for effi cient exploration of alternative application instances," in *CODES '02*, 2002, pp. 7–12.

[6] T. Stefanov, "Converting weakly dynamic programs to equivalent process network specifi cations," 2004, phD thesis, Leiden University.

[7] S. Meijer, H. Nikolov, and T. Stefanov, "On compile-time evaluation of process partitioning transformations for kahn process networks," in *CODES+ISSS '09*, 2009, pp. 31–40.

[8] S. "Meijer, H. Nikolov, and T. Stefanov, "Throughput modeling to evaluate process merging transformations in polyhedral process networks," in *DATE'10*, 2010, pp. 747–752.

[9] H. Nikolov, T. Stefanov, and E. Deprettere, "Systematic and automated multiprocessor system design, programming, and implementation," in *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 27, no. 3, 2008, pp. 542–555.

[10] M. I. Gordon et al, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *ASPLOS-XII*, 2006, pp. 151–162.

[11] H. Nikolov, T. Stefanov, and E. Deprettere, "Multi-processor system design with espam," in *CODES+ISSS '06*, 2006, pp. 211–216.

[12] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "Streamit: A language for streaming applications," in *CC '02*, 2002, pp. 179–196.

[13] M. I. Gordon et al, "A stream compiler for communication-exposed architectures," in *ASPLOS-X*, 2002, pp. 291–303.

[14] M. Kudlur et al, "Orchestrating the execution of stream programs on multicore platforms," *SIGPLAN Not.*, vol. 43, no. 6, pp. 114–124, 2008.

[15] C. Ostler et al, "An ilp formulation for system-level application mapping on network processor architectures," in *DATE '07*, 2007, pp. 99–104.

[16] P. M. Carpenter et al, "Mapping stream programs onto heterogeneous multiprocessor systems," in *CASES '09*, 2009, pp. 57–66.

[17] J. Pino et al, "A hierarchical multiprocessor scheduling framework for synchronous datafbw graphs," EECS Department, University of California, Berkeley, Tech. Rep. UCB/ERL M95/36, 1995.

[18] E. A. Lee et al, "Synchronous data fbw," in *Proceedings of the IEEE*, vol. 75, no. 9, September 1987, pp. 1235–1245.

[19] G. Bilsen et al, "Cyclo-static datafbw," *IEEE Transactions on signal processing*, vol. 44, no. 2, pp. 397–408, 1996.