Throughput Modeling to Evaluate Process Merging Transformations in Polyhedral Process Networks

Sjoerd Meijer, Hristo Nikolov, Todor Stefanov
Leiden Institute of Advanced Computer Science (LIACS), Leiden University
{smeijer,nikolov,stefanov}@liacs.nl

ABSTRACT

We use the polyhedral process network (PPN) model of computation to program embedded Multi-Processor Systems on Chip (MPSoCs) platforms. If a designer wants to reduce the number of processes in a network due to resource constraints, for example, then the process merging transformation can be used to achieve this. We present a compile-time approach to evaluate the system throughput of PPNs in order to select a merging candidate which gives a system throughput as close as possible to the original PPN. We show results for two experiments on the ESPAM platform prototyped on a Xilinx Virtex 2 Pro FPGA.

I. INTRODUCTION

The programming of 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. The `pn` compiler [1] has been developed to ease this task. It derives polyhedral process networks (PPNs), a specific case of Kahn Process Networks (KPNs), from sequential nested loop programs. Thus, autonomous processes and FIFO communication between different processes are automatically derived from sequential program specifications, that allow a natural mapping of processes to processing elements of the architecture [2]. In the `pn` partitioning strategy, a process is created for each function call statement in the nested loop program. As a result, the number of processes in the PPN is equal to the number of function call statements in the nested loop program. This partitioning strategy may not necessarily result in a PPN that meets the performance or resource requirements. To meet these requirements, a designer can apply algorithmic transformations to increase parallelism by unfolding processes [3] or to decrease parallelism by merging processes into a single component [4]. For the unfolding transformation, it has been shown in [5] that processes can be unfolded in many different ways which can result in significant differences in performance. Our current paper shows that the same holds for the process merging transformation: many solutions exist to merge different processes in a PPN with great differences in performance results and it's not trivial to select the best solution. Therefore, in this paper, we focus on the process merging transformation and present a compile-time solution to evaluate different merging alternatives.

A. Problem Definition and Paper Contribution

The process merging transformation reduces the number of processes in a PPN by sequentializing a selected number of processes in a single compound process. Thus, less processes need to be mapped on the platform's processing elements, at the price of possibly having less processes running in parallel. A designer needs to apply the process merging transformation in case *i)* the number of processes is larger than the number of processing elements, or *ii)* the network is not well balanced and therefore the same overall performance can be achieved using less resources. For both cases, the problem is that many different options exist to merge two or more processes. The total number of options to merge different processes for a PPN with *n* processes is $\sum_{i=2}^{n} \begin{pmatrix} n \\ i \end{pmatrix}$. To give an example for a PPN with 5 processes there are $\begin{pmatrix} 5 \\ 2 \end{pmatrix} + \begin{pmatrix} 5 \\ 3 \end{pmatrix} + \begin{pmatrix} 5 \\ 4 \end{pmatrix} + \begin{pmatrix} 5 \\ 5 \end{pmatrix} = 26$ different options to merge 2, 3, 4, or 5 processes. The challenge is how to find the best solution from all these options. Therefore, the main contribution of this paper is: the definition of an analytical throughput modeling framework for polyhedral process networks. It is used to evaluate the throughput of different process mergings in order to select the best option which gives a system throughput as close as possible to the original PPN.

B. Motivating Examples

With 3 motivating examples we show that selecting the best merging option is not a straightforward task as it depends on the inter-play of many factors which may not be evident at first sight. The first factor to be considered is the *workload* of a process. The workload W_{P_i} of a process P_i denotes the number of time units that are required to execute a function, i.e., the pure computational workload, excluding the communication. Figure 1 shows a PPN consisting of 6

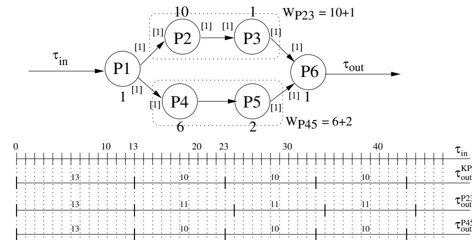

Fig. 1. Process Workload Influencing the System Throughput

processes. It is annotated with the process workload and

978-3-9810801-6-2/DATE10 © 2010 EDAA

shows the number of readings/writings from/to each FIFO channel. Process $P2$, for example, has a workload of 10 time units and a single token is read/written from/to a FIFO channel per process firing, which is denoted by "$[1]$" and can be repeated (possibly) infinitely many times. The network has two datapaths $DP1 = (P1, P2, P3, P6)$ and $DP2 = (P1, P4, P5, P6)$ that transfer an equal amount of tokens. We observe that process $P2$ determines the system throughput, which is illustrated with the time lines at the bottom of Figure 1. The first time line shows the rate τ_{in} at which tokens arrive at the network, i.e., each time unit. The second time line shows the system throughput of the original PPN, denoted by τ_{out}^{PPN}. Process $P6$ needs 13 time units (1+10+1+1) to produce its first token. Then, it produces a new token each 10 cycles which is dictated by the slowest process $P2$. If we apply the process merging transformation to processes $P2$ and $P3$, then compound process $P23$ becomes the most computationally intensive process of the network. Processes $P2$ (10 time units) and $P3$ (1 time unit) are sequentialized and thus it will take 10+1=11 time units instead of 10 time units for process $P6$ to produce a new token, as shown in the time line denoted by τ_{out}^{P23}. We observe that the throughput of this network is lower than the original PPN. The fourth time line, denoted by τ_{out}^{P45}, shows the system throughput after merging processes $P4$ and $P5$. In this case, however, we see that the system throughput is not affected, i.e., it is the same as the original PPN, because the two merged and sequentialized processes do not dictate the system throughput. Thus, a designer can safely merge these processes and achieve the same system throughput as the original PPN. With the following example, we show that considering the process workload W_{P_i} only is not enough; a second factor that needs to be taken into account is the *rate of producing tokens*. Consider

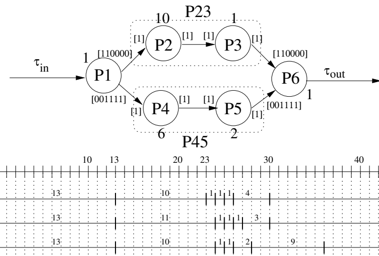

Fig. 2. Production Rate Influencing the System Throughput

the PPN in Figure 2 which is topologically the same as in the previous example. The only difference is that both datapaths transfer a different number of tokens. This is indicated with the patterns `[110000]` and `[001111]` 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 4. As a consequence of these patterns, more tokens are communicated through the second datapath $DP2$. Therefore, we observe that, despite process $P2$ largest workload of 10 time units, process $P4$ with a workload of 6 is more dominant. Therefore, merging processes $P4$ and $P5$ leads to a lower network throughput compared to merging $P2$ and $P3$, as can be seen in the time lines τ_{out}^{P45} and τ_{out}^{P23} in Figure 2. We observe a trend which is completely different from the previous example. According to Figure 2, a designer can safely merge processes $P2$ and $P3$ as opposed to $P4$ and $P5$ to achieve a system throughput that is equal to the original PPN. In the last motivating example, we consider the PPN shown in Figure 3. The processes always read and/or write a single token when they are fired. Therefore, one could expect that this example is different from the example in Figure 2, but similar as in Figure 1. We show, however, that neither case applies and that a third factor needs to be taken into account. In this example, process $P1$ is the computationally most intensive process with a workload of 53 time units. If a designer wants to merge processes, a logical choice would be to merge $P2$ and $P3$ and not to consider the heavy process $P1$. Processes $P2$ and $P3$ both have a workload of 25 time

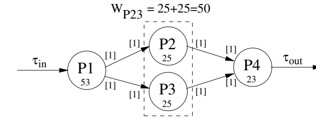

Fig. 3. Sequentialized FIFO Accesses Influencing the System Throughput

units and thus the compound process $P23$ has a summed workload of 50 time units, which is smaller than process $P1$ (53 time units). For this reason, we expect performance results that are equally good as the original PPN. However, when we measure the performance results of both the original PPN and the transformed PPN on the ESPAM platform [6], there is a 20% degradation in the performance results. Although the workload of compound process $P23$ is lower than $P1$, the compound process reads sequentially from two input channels and writes sequentially to two output channels. This makes it the heaviest process in the network. So, besides sequential execution of the process workloads, we observe that *sequential FIFO reading/writing* is another aspect that should be taken into account. The 3 examples above show that it is not trivial to merge processes and to achieve performance results as close as possible to the original PPN. Therefore, we want to have a compile-time framework to evaluate the system throughput such that the best possible merging can be selected.

C. Scope of Work

First of all, we consider acyclic PPN graphs. Cycles in a PPN are responsible for sequential execution of some of the processes involved in the cycle. The sequential execution can vary from a single initial delay, to a delay at each firing of some of the processes. For accurate throughput modeling, these cycles must be taken into account which we do not study in this work. Secondly, it is important to state that our goal is not to compare different PPNs, but to compare transformed PPNs derived from a single PPN. Therefore, in the throughput modeling, we do not take into account latencies. Thus, we do not calculate the total execution time of PPNs, but only want to capture the throughput trend instead. The reason is that

the framework should be fast, and only as accurate as needed to correctly capture the throughput trend for different process mergings. Thirdly, the process workload W_{P_i} is a parameter in our system throughput modeling. It should be provided by the designer who can obtain it, for example, by executing the function once on the target platform. Although our approach is extensible to heterogeneous MPSoCs, we restrict ourself to MPSoCs with homogeneous cores. Therefore, the process workload of a given process is a constant for all the cores in the platform. Finally, we do not study the effect of different buffer sizes. Although buffer sizes play an important role in the performance results, there are studies [7] showing that saturation points can be found where performance does not increase for larger buffer sizes. The pn compiler can find such points and we use buffer sizes that correspond to these points, i.e., the buffer sizes that give maximum performance.

II. Solution Approach

Before introducing the solution approach for throughput modeling, we first give some notations that are used throughout the rest of the section. We introduce the solution approach with an example, and then define all concepts in detail. Finally, we present the overall algorithm for the throughput modeling.

A. Notations

A function call statement has a number of input and output arguments and therefore the corresponding process in the PPN has a number of input/output ports to read/write data. Consequently, the structure of the process consists of a list of input ports, a function call, and a list of output ports. We refer to these networks as Polyhedral Process Networks (PPNs), as input/output ports and iteration domains are polytopes that can be efficiently manipulated and analyzed. The iteration space domain of a process P_k corresponds to all iterations of statement k in the nested loop program and is defined as $D_{P_k} = \{x \in \mathbb{Z}^d \mid Ax + b \geq 0\}$, where d is the depth of the loop nest. For process P in Figure 6, the iteration space domain is a two dimensional space defined as $D_P = \{(i,j) \in \mathbb{Z}^2 \mid 0 \leq i \leq 9 \wedge 0 \leq j \leq 9\}$. The n-th input port domain $IP^n_{P_k}$ of process P_k is defined as a subset of the process iteration space where data is read from an incoming FIFO channel: $IP^n_{P_k} \subseteq D_{P_k}$. Similarly, we define an output port domain $OP^n_{P_k} \subseteq D_{P_k}$ as the subset where data is written to an outgoing FIFO channel. In Figure 6, the FIFO read/write primitives are guarded by if-statements defining the input/output port domains. So, the first input port domain is defined as $IP^1_P = \{(i,j) \in \mathbb{Z}^2 \mid 0 \leq i,j \leq 9 \wedge i < 5\}$.

B. Process Throughput and Throughput Propagation

The solution approach for the overall Polyhedral Process Network (PPN) throughput modeling relies on calculating the throughput τ_{P_i} of a process P_i for all processes and propagation of the lowest process throughput to the sink nodes. For a process P_i, the propagation consists of selecting either the aggregated incoming FIFO throughput $\tau_{F_{aggr}}$ or the isolated process throughput $\tau^{iso}_{P_i}$:

$$\tau_{P_i} = min(\tau_{F_{aggr}}, \tau^{iso}_{P_i}), \quad (1)$$

Before defining formally $\tau_{F_{aggr}}$ and $\tau^{iso}_{P_i}$ (in Sections II-C - II-E), we first give an intuitive example of the solution approach applied on the PPN shown in Figure 3 and explain the meaning of Equation 1. First, the workload of each process is taken into account and let us assume that it takes 10, 20, 10, 10 time units for processes $P1, P2, P3, P4$, respectively, for firing its function. This means that, for example, $P1$ can read and produce a new token every 10 time units if there is input data. Thus, we define the isolated process throughput to be $\tau^{iso}_{P1} = \frac{1}{10}$ tokens per time units (excluding communication costs for the sake of simplicity). Similarly for the other processes, we define $\tau^{iso}_{P2} = \frac{1}{20}, \tau^{iso}_{P3} = \frac{1}{10}, \tau^{iso}_{P4} = \frac{1}{10}$. However, the required input data for a process can be delivered with a different throughput, i.e., the aggregated incoming FIFO throughput $\tau_{F_{aggr}}$. Consequently, the lowest throughput ($\tau_{F_{aggr}}$ or $\tau^{iso}_{P_i}$) determines the actual process throughput τ_{P_i}. Therefore, the minimum throughput value is selected as shown in Equation 1. This is repeated for all processes by iteratively applying Equation 1 on each process to select the lowest throughput and to propagate it to the sink processes. First, the PPN graph is topologically sorted to obtain a linear ordering of processes, i.e., $P1, P2, P3, P4$. In step I) of Figure 4, process

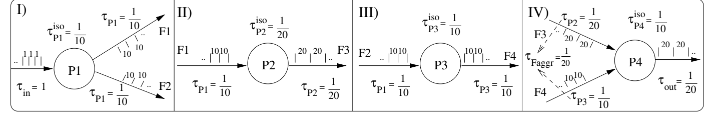

Fig. 4. Throughput Propagation Example

$P1$ is the first process to be considered. While it receives tokens at each time unit ($\tau_{in} = 1$), it is ready to fire again after 10 time units due to the process workload ($\tau^{iso}_{P1} = \frac{1}{10}$). We see that the actual process throughput is determined by the process itself (it is the slowest) and Equation 1 is used to find this: $\tau_{P1} = min(1, \frac{1}{10}) = \frac{1}{10}$ with which it writes to both its outgoing FIFO channels $F1$ and $F2$. If we continue with the second process in step II), we see that $P2$ receives tokens from $P1$ with a throughput of $\tau_{P1} = \frac{1}{10}$, however, $P2$ is twice slower than $P1$ which is delivering the data: $\tau_{P2} = min(\frac{1}{10}, \frac{1}{20}) = \frac{1}{20}$. Thus, we know that $P2$ writes its results to FIFO channel $F3$ with a throughput of $\frac{1}{20}$. For brevity, we skip $P3$ and consider the next process $P4$ in step IV). Process $P4$ reads from two FIFO channels $F3$ and $F4$, which are written by $P2$ and $P3$ with different throughputs. Therefore, the FIFO throughput must be aggregated in order to have a single throughput value at which data arrives. If we assume that both channels are read per firing of $P4$, then the slowest FIFO throughput determines the aggregated FIFO throughput. For this example, $\frac{1}{20}$ is the slowest component and we set $\tau_{F_{aggr}} = \frac{1}{20}$. Applying Equation 1 shows that the data is delivered with a lower throughput than $P4$ can actually process: $\tau_{P4} = min(\frac{1}{20}, \frac{1}{10}) = \frac{1}{20}$ and set this to be the process throughput. In this way, we have propagated the slowest throughput from $P2$ to the sink process $P4$, which in the end determines the system throughput. In the next sections we exactly define how all terms can be calculated.

C. Isolated Throughput of a (Compound) Process

The isolated throughput of a process P_i, denoted by $\tau_{P_i}^{iso}$, is the throughput of a process when it is completely isolated from its environment. This means that the isolated process throughput is determined only by the workload W_{P_i} of a process and the number of FIFO reads/writes per process firing provided that no blocking occurs:

$$\tau_{P_i}^{iso} = \frac{1}{W_{P_i} + x \cdot C^{Rd} + y \cdot C^{Wr}} = \frac{1}{C_{P_i}^{fire}}, \tag{2}$$

where x and y denote how many FIFOs are read and written per process firing and C^{Rd} and C^{Wr} the costs for reading/writing a single token from/to a FIFO channel. To the cost for a single process firing we refer as $C_{P_i}^{fire}$. It is important to note that two factors as identified in the motivating examples are taken into account here: the *process workload* by W_{P_i}, and the number of *sequential FIFO accesses* by x and y. As a function always needs values for its input arguments in order to fire, the number of FIFO reads x is equal to the number of function input arguments. This is different for writing output values as, for example, these can be broadcasted or written in some regular patterns. Therefore, we calculate an average number of FIFO writes y per iteration by summing all points in the output port domains and dividing this by the total domain size:

$$y = \frac{\sum_{i=1}^{n} |OP^i|}{|D_{P_j}|} \tag{3}$$

In a similar way, we must also model the firing cost for a compound process P_m in order to evaluate the system throughput for a PPN with merged processes. Assume that P_m is formed by merging processes P_i and P_j with iteration domains D_{P_i} and D_{P_j}, respectively. We define the isolated compound process throughput as $\tau_{P_m}^{iso} = \frac{1}{C_{P_m}^{fire}}$, where

$$C_{P_m}^{fire} = \frac{|D_{P_i}|}{|D_{P_j}|} \cdot (C_{P_i}^{fire} + C_{P_j}^{fire}) + \frac{|D_{P_j}| - |D_{P_i}|}{|D_{P_j}|} \cdot (C_{P_j}^{fire}) \tag{4}$$

with $|D_{P_i}| < |D_{P_j}|$. To model the cost $C_{P_m}^{fire}$ for firing the compound process, we take into account the generated schedule of the compound process as produced by the pn and ESPAM tools [6]. The execution of the process functions is interleaved as much as possible. This means that per firing of the compound process, all functions are sequentially executed if this allowed by the inter-process dependencies. In case of inter-process dependencies, an offset is calculated for the producer-consumer pair to ensure correct program behavior, and then the function execution is interleaved again. Therefore, we calculate fractions where the execution of the functions overlap and multiply it with the firing costs of these functions, i.e., the first term in Equation 4. And then we consider for the remaining firings the cost of the process with the largest domain size only, i.e., the second term in Equation 4. Note that the coefficients in Equation 4 represent these fractions which should sum up to 1. This formula can be generalized for any number of processes, but we omit it for the sake of brevity.

D. FIFO Channel Throughput

The throughput of a FIFO-channel is determined by the throughput of the processes accessing it. Let us consider the example shown in Figure 5. Assume that *P1* fires 500 times, i.e., $|D_{P1}| = 500$, and each time it writes to *F1* and *F2*.

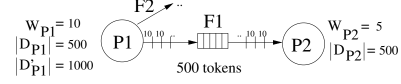

Fig. 5. FIFO Channel Throughput

Process $P1$ needs 10 time units to produce a token. Consumer process $P2$ is twice faster and needs only 5 time units to consume a token, but still it receives tokens only each 10 time units due to the slower producer. As a result, *P2* blocks on reading and waits for data, which follows the operational semantics of the PPN model of computation: a process stalls if it tries to read from an empty FIFO channel and proceeds only if data is available again. This example shows that, to calculate the FIFO throughput τ_{f_i} of a FIFO channel f_i, the minimum is taken of the FIFO write throughput $\tau_{f_i}^{Wr}$ and the FIFO read throughput $\tau_{f_i}^{Rd}$:

$$\tau_{f_i} = min(\tau_{f_i}^{Wr}, \tau_{f_i}^{Rd}), \tag{5}$$

where $\tau_{f_i}^{Wr} = \tau_{P1}$ (see Equation 1) and $\tau_{f_i}^{Rd} = \tau_{P2}^{iso}$ (see Equation 2). Let us consider another example where *P1* fires 1000 times, i.e., $|D'_{P1}| = 1000$ as also shown in Figure 5. Assume that in one iteration of *P1* data is written to FIFO channel *F1*, and in the next iteration to *F2*. This is repeated such that in total 500 tokens are written to both FIFOs *F1* and *F2*. To compensate for a producer that does not write data to a FIFO channel at each iteration, we define a coefficient that divides the total number of tokens transfered over a channel by the iteration domain size of a producer process P_i. This coefficient denotes an average production rate, expressed in a number of producer iteration points. Note that this takes into account the different *production rates* of processes as also identified in the motivating example in Figure 2. By multiplying this coefficient with the process throughput, we define FIFO write/read throughput $\tau_{f_i}^{Wr}$ and $\tau_{f_i}^{Rd}$ of a FIFO channel f_i as shown in Equations 6 and 7. In this way, we model a lower throughput if necessary.

$$\tau_{f_i}^{Wr} = \frac{|OP_{P_i}^j|}{|D_{P_i}|} \cdot \tau_{P_i} \quad (6) \qquad \tau_{f_i}^{Rd} = \frac{|IP_{P_i}^j|}{|D_{P_i}|} \cdot \tau_{P_i}^{iso}, \quad (7)$$

For the example, we see that $\tau_{f1}^{Wr} = \frac{500}{1000} \cdot \frac{1}{10} = \frac{1}{20}$ and the FIFO read throughput is $\tau_{f1}^{Rd} = \frac{500}{500} \cdot \frac{1}{5} = \frac{1}{5}$. Consequently, the FIFO throughput is $\tau_{f1} = min(\frac{1}{20}, \frac{1}{5}) = \frac{1}{20}$ tokens per time unit.

E. Aggregated FIFO Throughput

The throughput of a process τ_{P_i} is either determined by the FIFO throughput from which it receives its data, i.e., $\tau_{F_{aggr}}$,

or by the computational workload of the process itself, i.e., $\tau_{P_i}^{iso}$, as shown in Equation 1. $\tau_{P_i}^{iso}$ is computed with Equation 2. Here we show how to compute $\tau_{F_{aggr}}$. The throughput of the incoming FIFO channels are aggregated according to the way the function input arguments are read. Three examples are given at the right-hand side in Figure 6. Process P has **one**

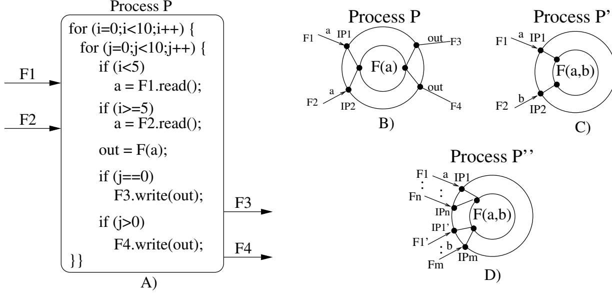

Fig. 6. Process Structure (left) and FIFO Throughput Aggregation (right)

input argument value a that is read from two different input ports $IP1$ and $IP2$. Thus, two tokens are delivered, but only one is read for each firing of the consumer process. The other token will be read in another firing. To model the throughput at which data arrives, the sum is taken of the FIFO throughput $F1$ and $F2$. Effectively, this means that the aggregated incoming FIFO throughput becomes higher, which corresponds to the behavior that one token is needed but two are delivered. Note that any imbalance in the number of tokens transfered over each FIFO channel has already been taken into account in the FIFO read/write throughput as defined in Equation 6 and 7. Process P' is the second example, which reads its **two** input arguments values a and b from FIFOs $F1$ and $F2$. Thus, both FIFOs are read per firing of P'. If one FIFO throughput is fast and the other one is slower, then the slowest FIFO throughput determines the aggregated FIFO throughput and, in such cases, the minimum throughput is taken. Thirdly, the general case is illustrated with process P'' and combines the previous two examples. It has multiple function input arguments and multiple incoming FIFO channels per input argument. The aggregated FIFO throughput $\tau_{F_{aggr}}$ is calculated by taking the sum per function input argument, and then the minimum of these values denotes the aggregated FIFO throughput:

$$\tau_{F_{aggr}} = min(\sum_{i}^{n} \tau_{f_i}^{Rd}, ..., \sum_{j}^{m} \tau_{f_j}^{Rd}). \quad (8)$$

Up to now, we have formally defined all the components that allow the throughput calculation and propagation to be done in a systematic and automated way. The pseudo code of the throughput calculation and propagation algorithm is shown in Algorithm 1. This algorithm was explained with the example in Section II-B.

Algorithm 1 : PPN Throughput Estimation Pseudo-code

Require: PPN: a Polyhedral Process Network
Require: W_{P_i}: the computational workload of all processes.
$list \leftarrow$ Create topological ordering for PPN
for all process $P_i \in list$ **do**
1) $\tau_{P_i}^{iso} = set_isolated_throughput(P_i, W_{P_i})$
2) Set $\tau_{f_j}^{Rd}$ for all incoming FIFOs f_j and $\tau_{F_{aggr}} = calc_fifo_aggr(\tau_{f_j}^{Rd}, .., \tau_{fn}^{Rd})$
3) $\tau_{P_i} = min(\tau_{P_i}^{iso}, \tau_{F_{aggr}})$
4) Set $\tau_{f_j}^{Wr}$ for all outgoing FIFO f_j of P_i.
end for
return $\tau_{out}^{PPN} = \tau_{sink}$

III. Experiments and Results

In this section we map two different nested loop kernels on the ESPAM platform prototyped on a Xilinx Virtex 2 Pro FPGA. Each process is mapped one-to-one on a MicroBlaze softcore processor and the processors are point-to-point connected. FIFO communication is implemented with FSL links and a FIFO access costs 7 clock cycles. We investigate if our throughput modeling captures the differences in performance results for different process merging configurations and process workloads.

A. Experiment I

We merge two light-weight producers (workload of 54 time units) into a single process in this experiment and should observe that the new compound process does not become the process that dictates the system throughput. Then, we increase the workload of the producers to 59 time units to achieve the opposite and test whether this is captured by the model. Figure 7 shows the nested loop program in A), the derived

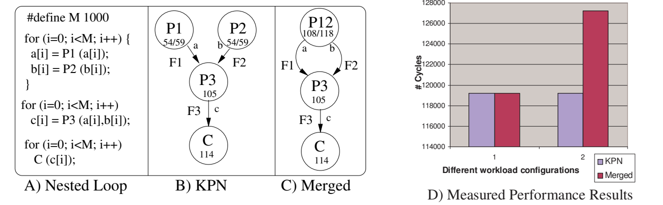

Fig. 7. PPNs and Performance Results on the ESPAM Platform

PPN in B), and the PPN with producers $P1$ and $P2$ merged in C). We calculate the throughput of the PPN before and after merging by applying Algorithm 1.

0 $list = \{P1, P2, P3, C\}$
1.1 $\tau_{P1}^{iso} = \frac{1}{54+0+7} = \frac{1}{61}$
1.2 $\{\emptyset\} = get_in_fifos(P1)$
1.3 $\tau_{P1} = min(\frac{1}{61}, \infty) = \frac{1}{61}$
1.4 $\{F1\} = get_out_fifos(P1)$
1.4.1 $\tau_{F1}^{Wr} = \frac{1000}{1000} \cdot \frac{1}{61} = \frac{1}{61}$
:
3.1 $\tau_{P3}^{iso} = \frac{1}{105+(2\cdot7)+7} = \frac{1}{126}$
3.2 $\{F1, F2\} = get_in_fifos(P3)$
3.2.1 $\tau_{F1}^{Rd} = \frac{1000}{1000} \cdot \frac{1}{126}$
3.2.2 $\tau_{F2}^{Rd} = \frac{1000}{1000} \cdot \frac{1}{126}$
3.2.3 $\tau_{F1} = min(\tau_{F1}^{Wr}, \tau_{F1}^{Rd})$
3.2.3 $\tau_{F1} = min(\frac{1}{61}, \frac{1}{126}) = \frac{1}{126}$
3.2.4 $\tau_{F2} = min(\frac{1}{61}, \frac{1}{126}) = \frac{1}{126}$
3.2.5 $\tau_{F_{aggr}} = min(\frac{1}{126}, \frac{1}{126}) = \frac{1}{126}$
3.3 $\tau_{P3} = min(\frac{1}{126}, \frac{1}{126}) = \frac{1}{126}$
3.4 $\{F3\} = get_out_fifos(P3)$
3.4.1 $\tau_{F3}^{Wr} = \frac{1000}{1000} \cdot \frac{1}{126} = \frac{1}{126}$
4.1 $\tau_{C}^{iso} = \frac{1}{114+7+0} = \frac{1}{121}$
4.2 $\{F3\} = get_in_fifos(C)$
4.2.1 $\tau_{F3}^{Rd} = \frac{1000}{1000} \cdot \frac{1}{121} = \frac{1}{121}$
4.2.2 $\tau_{F3} = min(\frac{1}{126}, \frac{1}{121}) = \frac{1}{126}$
4.3 $\tau_{C} = min(\frac{1}{126}, \frac{1}{121}) = \frac{1}{126}$
5 $\tau_{out}^{PPN} = \tau_{C} = \frac{1}{126}$

Fig. 8. Throughput Estimation of Processes $P1, P3, P4$

Figure 8 shows the analysis for process $P1$, $P3$ and $P4$. We omit the analysis for $P2$ as it similar to $P1$. In process $P3$, two FIFO throughput values are aggregated as shown in step 3.2.5. We find a process throughput of $\tau_{P3} = \frac{1}{126}$ for process

$P3$, which is propagated to C such that the system throughput is $\tau_{out}^{PPN} = \tau_C = \frac{1}{126}$ as well. When we compute the system throughput for the PPN with processes $P1$ and $P2$ merged, we find a system throughput of $\tau_{out}^{PPN'} = \frac{1}{126}$. This means that we predict that the original PPN and transformed PPN′ perform equally well. This is confirmed by the actual measured performance results shown in Figure 7 D). The first bar denotes the cycle numbers for the original PPN, and the second bar for the transformed PPN′. Then we increase the workload of the producer processes and intentionally create a compound process that is the most compute intensive process and check if this is captured by our throughput model. When we analyze the throughput we find $\frac{1}{126}$ and $\frac{1}{152}$ for the original and transformed PPN, respectively. The calculation indicates that the throughput of the merged PPN is lower, which is confirmed by the third and fourth bar in the measured performance results in Figure 7 D).

B. Experiment II

In this experiment we consider a more complicated network shown in Figure 9 that combines different properties. First of all, it has processes with different domain sizes. Processes $P1$ and $P2$ fire 500 times, while the other processes fire 1000 times. As a result, coefficients will scale down the $F1$ and $F2$ FIFO read throughput. Secondly, two data paths come together in process $P3$ where one token is needed per firing of $P3$ similar to the example in Figure 6 B). Thirdly, in process $P6$ two datapaths are joined as well where both tokens are needed for each firing, similar to the example in Figure 6 C). We estimate the system throughput by applying Algorithm 1

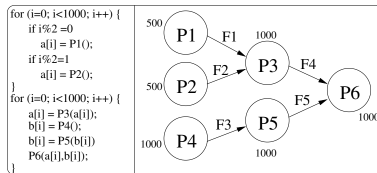

Fig. 9. Nested-loop Program and its Derived PPN

again and test the throughput modeling with 3 different process workload configurations. Each configuration is a tuple where the first value corresponds to the workload of process P1, the 2nd value to workload of P2, etc. Figure 10 shows the measured performance results and for each configuration the original PPN in Figure 9 is used as a reference (the first bar) and different mergings are shown in the 2nd, 3rd and 4th bars. For example, the second bar denotes the performance results after merging processes P1, P2 and P3. If we take the 2nd workload configuration as an example, our model finds the following throughputs: $\frac{1}{85}, \frac{1}{121}, \frac{1}{85}, \frac{1}{115}, \frac{1}{113}$. Thus, the estimation indicates that the first merging ($\frac{1}{121}$), leads to a lower throughput than the original PPN ($\frac{1}{85}$). The second merging ($\frac{1}{85}$) gives the same performance results, and the third ($\frac{1}{115}$) and fourth ($\frac{1}{113}$) are worse than the original PPN. From these estimations, we conclude that processes $P3$ and $P4$ can be merged and achieve the same system throughput. This estimation is correct as confirmed by the actual measured performance results shown in Figure 10.

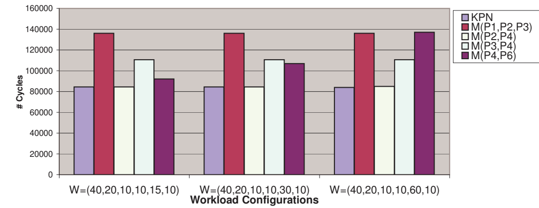

Fig. 10. Measured Results on the ESPAM Platform

IV. Related Work

While many works focus on the code generation of clustered or grouped tasks itself, as it often called in the domain of Synchronous Data Flow (SDF) graphs [8], we analyze and model networks with a given compound process and schedule to compare different PPN instances. There are other works on throughput computation, but they are developed for SDF and CSDF models [9], [10]. An approach is presented in [11] to automatically synthesize a multiprocessor architecture for process networks under particular mapping and performance constraints. This is different from our work, as the process networks are not analyzed.

V. Conclusion

We have presented a solution approach for throuhgput modeling of polyhedral process networks to evaluate process merging transformations. Our approach takes into account all factors that influence the throughput. Therefore, we can accurately capture the throughput trend and select the best possible merging as illustrated with the experiments.

References

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

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

[3] T. Stefanov, B. Kienhuis, and E. Deprettere, "Algorithmic transformation techniques for efficient exploration of alternative application instances," in *Proc. of CODES*, 2002, pp. 7–12.

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

[5] S. Meijer, H. Nikolov, and T. Stefanov, "On compile-time evaluation of process partitioning transformations for kahn process networks," in *Proc. of CODES+ISSS*, 2009.

[6] H. Nikolov, T. Stefanov, and E. Deprettere, "Systematic and automated multiprocessor system design, programming, and implementation," *IEEE TCAD*, vol. 27, no. 3, pp. 542–555, 2008.

[7] E. Cheung, H. Hsieh, and F. Balarin, "Automatic buffer sizing for rate-constrained kpn applications on multiprocessor system-on-chip," in *Proc. of HLDVT*, 2007, pp. 37–44.

[8] J. Falk, J. Keinert, C. Haubelt, J. Teich, and S. S. Bhattacharyya, "A generalized static data flow clustering algorithm for mpsoc scheduling of multimedia applications," in *Proc. of EMSOFT*, 2008, pp. 189–198.

[9] A. H. Ghamarian et. al., "Parametric throughput analysis of synchronous data flow graphs," in *Proc. of DATE*, 2008, pp. 116–121.

[10] A. Moonen et. al., "Practical and accurate throughput analysis with the cyclo static dataflow model," in *Proc. of MASCOTS*, 2007, pp. 238–245.

[11] B. K. Dwivedi, A. Kumar, and M. Balakrishnan, "Automatic synthesis of system on chip multiprocessor architectures for process networks," in *Proc. of CODES+ISSS*, 2004, pp. 60–65.