Universiteit
Leiden
The Netherlands

Semi-partitioned scheduling and task migration in dataflow networks
Cannella, E.

Citation
Cannella, E. (2016, October 11). *Semi-partitioned scheduling and task migration in dataflow networks*. Retrieved from https://hdl.handle.net/1887/43469

Version:	Not Applicable (or Unknown)
License:	Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden
Downloaded from:	https://hdl.handle.net/1887/43469

Note: To cite this publication please use the final published version (if applicable).

Cover Page

Universiteit Leiden

The handle http://hdl.handle.net/1887/43469 holds various files of this Leiden University dissertation

Author: Cannella, Emanuele
Title: Semi-partitioned scheduling and task migration in dataflow networks
Issue Date: 2016-10-11

Chapter 3

PPN Communication on Networks-on-chip

Most of the work presented in this chapter has been published in [CDS11].

IN this chapter and in the following one, Chapter 4, we present techniques which are aimed at achieving system adaptivity[1] in the context of **best-effort systems**. In order to make the system adaptive, our approach provides a mechanism by which application processes can migrate among processors at run-time.

Our approach takes into account the emerging trends in the design of embedded MPSoCs that are described in Sections 1.1.1 and 1.1.2. That is, we base our technique on the following two assumptions:

1. As the methodology used to specify applications is concerned, we consider an approach based on a Model of Computation. In particular, we adopt the PPN MoC, which is presented in Section 2.1.3. In PPNs, memory, control, and synchronization are completely distributed, which allows to change the mapping of processes to PEs at run-time with minor effort.
2. Regarding the choice of communication infrastructures, we assume that PEs in our systems are interconnected by a Network-on-Chip (NoC). Some of the advantages of NoCs are described in Section 1.1.2. In the context of system adaptivity, in particular, we argue that NoCs are appropriate because NoCs are generic, i.e., the same NoC-based platform can be used to run different applications or to run the same application with a different mapping of processes. As mentioned in Section 1.1.2, we consider NoC-based platforms which comprise several processing elements, organized in *tiles*.

From the above discussion, it follows that the PPN MoC and NoC-based interconnections, among other advantages, can favor system adaptivity in embedded MPSoCs.

[1] Recall the explanation of our understanding of the term "system adaptivity", given in Section 1.2.1 on page 10.

However, there is a mismatch between the communication primitives allowed in NoC-based execution platforms and the semantics of the PPN MoC. Therefore, in this chapter we investigate and propose several approaches to overcome this mismatch. All of the proposed approaches are aimed at implementing PPN communication on NoCs considering system adaptivity as a driving objective. Moreover, they do not require specific hardware support from the NoC-based platform to realize inter-tile communication among PPN processes. The approaches presented in this chapter represent different possible implementations of the first component of the middleware layer that is proposed in this thesis (see Figure 1.6 on page 20) to achieve system adaptivity on NoC-based MPSoCs.

The remainder of this chapter is organized as follows. Section 3.1 continues the introduction by stating the addressed research problem. A summary of the contributions of this chapter and a list of related work is provided in Sections 3.2 and 3.3, respectively. The proposed and investigated approaches for PPN communication on NoCs are described in detail in Section 3.4. The applications used to evaluate the different approaches are explained in Section 3.5 followed by the performance results in Section 3.6. Note that in the rest of this chapter, for the sake of brevity, we will refer to an "approach for implementing PPN communication on NoCs" as a "PPN communication approach".

3.1 Problem Statement

The main problem addressed in this chapter is the implementation of an efficient approach to implement PPN communication on Network-on-Chip platforms. The first requirement that we consider is that this approach must respect the PPN communication semantics (recall Section 2.1.3). That is, processes must *block on read*, when trying to get data tokens from an empty FIFO, and *block on write*, when trying to write data tokens to a full FIFO. Moreover, we want our communication approach to be application-independent and oriented to system adaptivity.

The communication and synchronization problem when mapping PPNs on a NoC is depicted in Fig. 3.1, showing a producer P and a consumer C connected through a communication FIFO buffer B. We denote the size of buffer B as $size(B)$. Unless otherwise specified, throughout this thesis we will express the size of buffers in number of tokens. If both producer and consumer can directly access the status register of this FIFO buffer, to check if it is empty or full, implementing the PPN semantics is straightforward. However, in this chapter we consider NoC implementations with no direct remote memory access, that is, those NoCs in which a processing element has direct access only to the local memory of its tile. In this scenario, processes P and C, if mapped onto different tiles, cannot access the same piece of memory and they can only exchange tokens through the network. Thus, the FIFO buffer B has to be split on the producer tile and/or on the consumer tile. We denote the buffer allocated on the producer tile and consumer tile as B^P and B^C, respectively. Note that, as will be shown in one of our proposed PPN communication approaches (see Section 3.4.1), it is not necessary that both these buffers actually exist. However, in

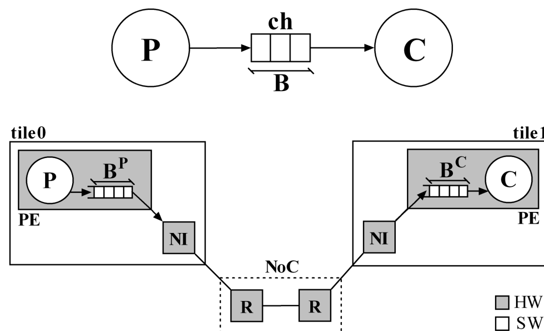


Figure 3.1: *The top part of the figure illustrates a producer-consumer pair which communicates through FIFO buffer B. The bottom part of the figure shows how this pair of processes can be mapped onto a NoC-based platform. The FIFO buffer B has to be split on the producer tile and/or on the consumer tile using two software FIFOs, namely B^P and B^C. Note that the approach presented in this chapter is independent of the considered NoC structure.*

general, if $size(B)$ is the minimum buffer size that guarantees deadlock-free execution of the original PPN graph, the size of B^P and B^C must be necessarily set such that $size(B^P) + size(B^C) \geq size(B)$. In this expression, if either B^P or B^C does not exist, its size is set to zero.

We aim at implementing the PPN semantics without a dedicated support from the underlying hardware architecture that allows checking for the status of the remote FIFO buffers. For instance, in Figure 3.1, process P cannot check the status of B^C, and process C cannot check the status of B^P. Moreover, we do not require support for multiple hardware FIFOs on each NoC tile. Each tile is endowed with only two hardware FIFOs[2], one for incoming messages and one for outgoing messages, both of which reside in the Network Interface (NI). However, we rely on the ability to transfer data, in both directions, from these hardware FIFO buffers to the *software FIFOs* (e.g., B^P and B^C in Figure 3.1) which implement the channels of our PPN and are accessed by the PPN processes.

As the consumer can access the status of B^C, implementing the *blocking read* is trivial because every time C wants to access B^C and this buffer is empty, the consumer just has to wait until tokens arrive from the producer tile. However, since the producer can only access the status of B^P, implementing the *blocking on write* behavior is more difficult. The producer must know that the remote buffer B^C is not full before sending tokens to C over the NoC. Several techniques can be considered to inform the producer about the status of the buffer on the consumer side. We compare the communication approaches that we have investigated in Section 3.4.

As a final requirement of our proposed techniques, we demand that our PPN communication approaches allow processes to be mapped on any tile of the system, without the need to change the actual code structure of the PPN application processes.

[2]These hardware FIFOs are not shown in Figure 3.1 to avoid clutter.

An example of such structure is shown in Fig. 2.3(b) on page 29. In particular, we want the communication primitives of PPN processes (*READ*, *WRITE*) to remain generic, without the notion of process mapping or hardware platform primitives. At run-time, these generic PPN primitives are then converted by the PPN communication approaches to corresponding hardware platform primitives which take into account the actual mapping of processes in the system. This is because the mapping of processes in the system can change due to a task migration.

3.2 Contributions

The contribution of this chapter is two-fold. First, we propose different PPN communication approaches that allow applications specified as PPNs to be executed efficiently on NoC-based platforms. The PPN communication approaches support any possible mapping of PPN processes in the system. Second, we ensure that these PPN communication approaches allow the run-time remapping capability of processes among the tiles of the NoC, thus enabling system adaptivity as considered in this thesis.

3.3 Related Work

Kahn Process Networks (KPNs) [Kah74] is a widely studied model of computation used to specify concurrent stream-based applications. The KPN MoC is a superset of the PPN MoC considered in this chapter, therefore in the following paragraphs we list some works, which target KPNs, that are related to the problem addressed in this chapter.

Previous research on the use of KPNs in multiprocessor embedded systems has mainly focused on the design of frameworks which employ that MoC as a model for application specification [NSD08, NKG+02, KKJ+08], and which aim at supporting and optimizing the mapping of KPN processes on the nodes of a reference platform [BHHT10, HSH+09]. In [NSD08, NKG+02], different methods and tools are proposed to generate, in an automated way, KPN application specifications from sequential programs written in C/C++. Design space exploration tools and performance analysis are then usually employed for optimizing the mapping of the generated KPN processes on a reference platform. Then, in the successive design phase, software synthesis for multi-processor systems [KKJ+08, HSH+09] and/or architecture synthesis for FPGA platforms [NSD08] is performed.

The approaches described above, which map applications modeled as KPNs to hardware platforms tailored to the application KPN specification, have a strong coupling between the application and the hardware platform. Running a different application on the generated platform would not be possible or, even if possible, would give bad performance results. In this chapter, we adopt a different approach because we start with the assumption that we have a platform equipped with homogeneous cores well interconnected with a NoC. We provide a PPN API for this platform which implements inter-tile communication among PPN processes. Most importantly, the

PPN processes' code remains the same in all possible mappings of the processes. This is achieved by the proposed PPN communication approaches, that convert the generic PPN communication primitives to corresponding hardware platform primitives that follow the actual mapping of processes in the system.

This approach, where software synthesis relies on the high level APIs provided by the reference platform for facilitating the programming of a multiprocessor system, can be seen in other works in the literature. In fact, the trend from single core design to many core design has forced the research community to consider inter-processor communication issues for transferring data among the cores. One of the emerged message passing communication API is Multicore Association's Communication API (MCAPI) [MCA] that targets the inter-core communication in a multicore chip. MCAPI is the light-weight (low communication latencies and memory footprint) implementation of message passing interface APIs such as Open MPI [ope]. However, these MPI standards do not allow an efficient implementation of KPN (or PPN) semantics [DDF11] because building these semantics on top of their primitives incurs an additional overhead that may be disadvantageous.

The communication and synchronization problem when implementing KPNs on multi-processor platforms without hardware support for FIFO buffers has been considered in [NMSD09] and [HSH$^+$09]. In [NMSD09] the *receiver-initiated* method has been proposed and evaluated for the Cell BE platform. On the same hardware platform, [HSH$^+$09] proposes a different protocol, which makes use of mailboxes and *windowed FIFOs*. The difference with our work presented in this chapter is that we actually compare a number of approaches to implement the KPN semantics, and that we deal with a different kind of platform, with no Direct Memory Access support.

In [DDF11] the active *virtual connector* approach has been proposed and evaluated analytically, whereas our results are obtained by experiments on a real implementation. Moreover, in this chapter we propose yet another approach, namely *virtual connector with variable rate*.

The authors in [NGWK09] address the problem of implementing the KPN semantics on a NoC. However, in their approach the NoC topology is customized to the needs of the application at design time and network end-to-end flow control is used to implement the blocking write feature. In [NGWK09] no run-time task remapping is allowed, because the hardware platform is generated assuming a specific (fixed) mapping of KPN tasks. By contrast, in our work the PPN communication approaches allow run-time remapping of processes and, in turn, system adaptivity.

An approach to guarantee blocking write behavior for KPN processes on NoCs is also used in [Gab09]. In that work, a FIFO buffer that implements a KPN channel is allocated on the tile of the *consumer* process. Then, before sending data tokens, the producer process uses a dedicated operating system communication primitive which guarantees that the remote FIFO buffer is not full. Compared to this kind of protocol, the communication approaches described in this chapter assume a more active behavior of the consumer processes to guarantee the blocking on write behavior. That is, in our approaches the consumer process actively sends back to the producer some messages to inform the producer about the status of the remote FIFO buffer. We actually propose and evaluate three kinds of communication approaches, which re-

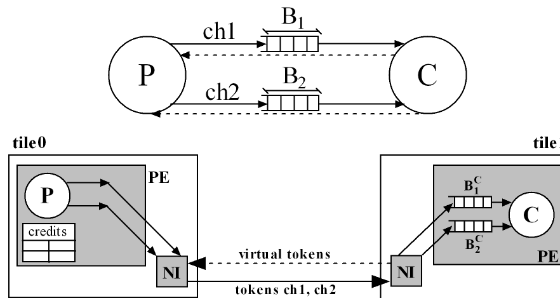


Figure 3.2: *Producer-consumer pair using the virtual connector method. Compared to Figure 3.1, notice that the producer tile does not contain any software FIFO for the considered channels. However, the producer process P uses a credit variable for each channel to keep track of the status of the FIFOs residing on the consumer tile.*

quire the consumer process to be active to a different extent (in terms of the amount of messages sent back to the producer process). The experimental results of Section 3.6.1 show that the communication approach which requires the most proactive behavior of the consumer achieves higher performance compared to the others.

3.4 PPN Communication Approaches

This section presents the different approaches that we have explored for the implementation of PPN processes communication and synchronization on a tiled NoC-based hardware platform. Three PPN communication approaches are proposed and investigated: Virtual Connector approach (VC), Virtual Connector with Variable Rate approach (VRVC), and Request-driven approach (R). Basically, the proposed approaches differ in the frequency of acknowledgment messages sent from the consumer process to the producer process regarding the status of the consumer FIFO buffers.

In all of the approaches described in what follows, system adaptivity is taken into account by using dedicated tables that list, among other information, the current[3] mapping of producer and consumer processes for each channel of the PPN graph. We refer to such tables as *middleware tables*. The current mapping of producer and consumer processes is checked when the PPN primitives (i.e., *READ, WRITE* in Figure 2.3(b) on page 29) are converted to the corresponding hardware platform primitives, such that tokens and synchronization messages are sent to the right tiles in the system. The middleware tables can be updated at run-time, ensuring correct communication in case of remapping of processes.

3.4.1 Virtual Connector approach (VC)

In the Virtual Connector approach, which is depicted in Fig. 3.2, for each channel in the original PPN graph we add a virtual[4] one in the opposite direction. This virtual connector is used for acknowledging the producer about the status of the FIFO buffer on the consumer tile. We adapted this approach, previously proposed in [DDF11], to the needs of our system implementation. In [DDF11], the proposed communication approach is *active*, meaning that it is implemented using separate threads which deal with the PPN communication, while our approach is *static*, with no separate threads dedicated to communication. Although a comparison of the static and active implementations may be worthwhile to do, in this chapter we adopt the static approach with the argument that the scheduling and synchronization of an additional thread dedicated to PPN communication will introduce an additional overhead due to the scheduling and context switching times.

For each channel in the original PPN graph we instantiate a software FIFO buffer on the consumer tile. The size of this buffer is set to the value of the original buffer size in the PPN graph. On the producer tile there are no software FIFOs when using this approach because tokens can be directly sent over the network via the NI. The PPN *blocking write* behavior is implemented by using a credit-system which guarantees that enough locations are free in the FIFO buffers of the consumer processes. Therefore, referring back to Fig. 3.1, in this approach for each channel i, $size(B_i^C) = size(B_i)$ and $size(B_i^P) = 0$.

In our implementation, we store on the producer side a variable for each channel, called *credit*, which represents the number of free slots in the remote FIFO buffer implementing that channel. At startup, the credit is set to the size of the remote FIFO ($credit_i = size(B_i^C)$), because all of its slots are free[5]. For each token sent over the network by the producer, the credit of the corresponding channel is decreased by one. The producer is allowed to send tokens over the network only if the credit is positive, otherwise it blocks. This implements the *blocking write* behavior. At the consumer side, for every token consumed from that channel, a virtual token (VT) is sent back to the producer via the virtual connector. For every virtual token received on the producer tile, the credit of the corresponding channel is increased by one. This way the producer is constantly updated about the status of the remote FIFO buffers.

Read and Write communication primitives

The read and write primitives use an auxiliary function called *process_NI_msgs()*. This function is used in the read primitive when the calling process is blocked on read, and in the write primitive when it is blocked on write. The *process_NI_msgs()* function checks the status of the NI buffer for incoming messages. If the buffer is not empty, it

[3] Although a task migration mechanism is not provided in this chapter, our proposed PPN communication approaches must allow the spatial mapping of tasks to change at run-time.

[4] These channels are said to be virtual because they are not used to communicate actual data.

[5] This holds unless there are initial tokens in B^C. In such a case, the value of $credit_i$ is decreased by the number of initial tokens.

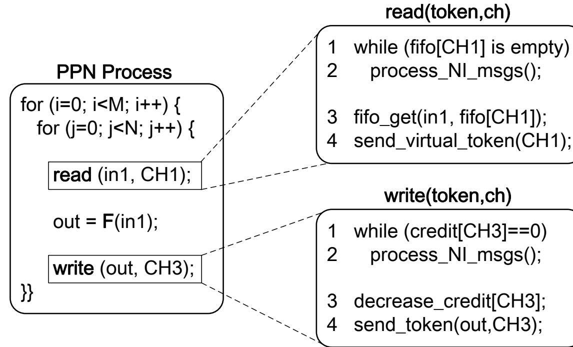


Figure 3.3: *Pseudocode of the VC approach. The left part of the figure shows an example of structure of a PPN process. The right side provides the pseudocodes of read and write PPN primitives as implemented in the VC communication approach.*

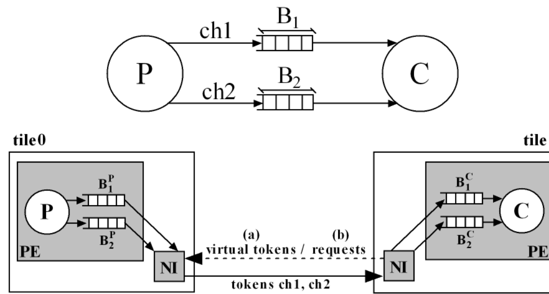


Figure 3.4: *Producer-consumer implementation: when using the VRVC approach, the producer receives back virtual tokens (a); when using the R approach, it receives requests (b).*

processes one message at a time, until all the incoming messages are consumed, in the following way. If the message is an incoming token for channel *i*, it stores the token in the software FIFO which implements channel *i*. If, instead, it is a virtual token for channel *j*, it consumes the token and increases the credit of channel *j*.

Read primitive. In the VC approach, the *read* primitive (used to read a token from channel *ch*) performs the following sequence of actions.

1. It checks if the FIFO buffer corresponding to *ch* contains data tokens (blocking read behavior). If the FIFO is empty, it keeps executing the auxiliary function *process_NI_msgs()* until the FIFO is no longer empty.
2. At this point, the FIFO corresponding to *ch* contains data tokens. Then, the read primitive gets a token from the FIFO.
3. Finally, a virtual token is sent back to the consumer process to acknowledge that a token has been read from the FIFO.

These actions are implemented in the *read* primitive in Fig. 3.3. Lines 1-2 implement the blocking read. If the FIFO buffer corresponding to the calling channel (in

the example, *CH1*) is empty, *process_NI_msgs()* is executed until new tokens for that channel reach the NI input buffer. Lines 3 and 4 complete the *read* primitive: the token is transferred from the software FIFO to *in1*, and a virtual token is sent back to the producer side of *CH1*. This is actually performed by putting in the NI outgoing buffer a message representing a virtual token for channel *CH1*.

Write primitive. In the VC approach, the *write* primitive (used to write a token to channel *ch*) performs the following sequence of actions.

1. It checks if the *credit* corresponding to channel *ch* is equal to zero (blocking write). In this case, it keeps executing the auxiliary function *process_NI_msgs()* until the the credit is no longer zero.
2. At this point, the credit corresponding to *ch* is greater than zero. Then, the credit is decreased by one to consider the fact that in the next step a token will be sent, over the NoC, to the consumer.
3. Finally, the token is sent to the consumer over the NoC.

These actions are implemented in the *write* primitive in Fig. 3.3. Lines 1-2 implement the blocking write behavior. If the credit is zero, *process_NI_msgs()* is executed. If virtual tokens for the blocked channel are received, the credit is then increased and this condition unblocks the write to that channel. Lines 3-4 complete the *write* procedure. The credit for the considered channel is decreased, and the token is sent over the network, which is done by putting in the NI outgoing buffer a message representing this token, and then letting the NI to perform the actual transfer over the NoC[6].

3.4.2 Virtual Connector with Variable Rate approach (VRVC)

This approach represents a variant of the *virtual connector* described in Section 3.4.1. The basic idea is that instead of sending one virtual token to the producer for *every* token consumed from channel *i*, the consumer sends it after n_i consumed tokens, where n_i is a parameter that can be set such that $1 \leq n_i \leq size(B_i)$, where $size(B_i)$ is the buffer size in the original PPN graph. The *credit* variable for channel *i* will then be increased by n_i for every virtual token received for that channel. This approach leads to a reduced traffic on virtual connectors, which can be beneficial in NoC implementations to avoid congestion of messages.

Since the sending back of virtual tokens does not happen for every consumed token, in some cases the PPN graph properties require to store, also at the producer side, tokens for the channels in order to avoid deadlocks. We provide an explanation of this phenomenon in Example 3.4.1.

Example 3.4.1. Consider the scenario depicted in Figure 3.5, where producer process P is connected to consumer process C through a channel *ch*, implemented by a FIFO buffer B. Assume that, for channel *ch*, the parameter n of the VRVC approach is set to $size(B)$, the size of buffer B in the original PPN graph. As mentioned earlier, FIFO buffers are required both on the producer and on the consumer tile. We denote these FIFO buffers as B^P and B^C, respectively. Assume that the size of these buffers are

[6]For a brief description of how messages are sent over the NoC, please refer to Section 1.1.2.

$size(B^P) = (size(B) - 1)$ and $size(B^C) = size(B)$. We will eventually show that this assumption is necessary.

We recall that, in the original PPN graph, the size of a certain FIFO buffer B is computed by the `pn` compiler [VNS07] such that deadlocks cannot occur due to a lack of space in B. The size of buffer B derived by `pn` is minimum, that is, at run-time it may happen that B is required to store a number of token equal to its size to avoid a deadlock. Therefore, since FIFO buffer B in the VRVC approach is split over B^P and B^C, in order to avoid deadlocks it is necessary that at any time up to $size(B)$ tokens can be stored over B^P and B^C. Denoting the number of tokens stored in buffer B^P and B^C at time instant t as $tkns(B^P, t)$ and $tkns(B^C, t)$, respectively, we have that, over time, $\big(tkns(B^P, t) + tkns(B^C, t)\big)$ can grow up to $size(B)$.

At run-time, the communication between producer process P and consumer process C may follow the sequence of macro-steps shown in Figure 3.5 and explained below.

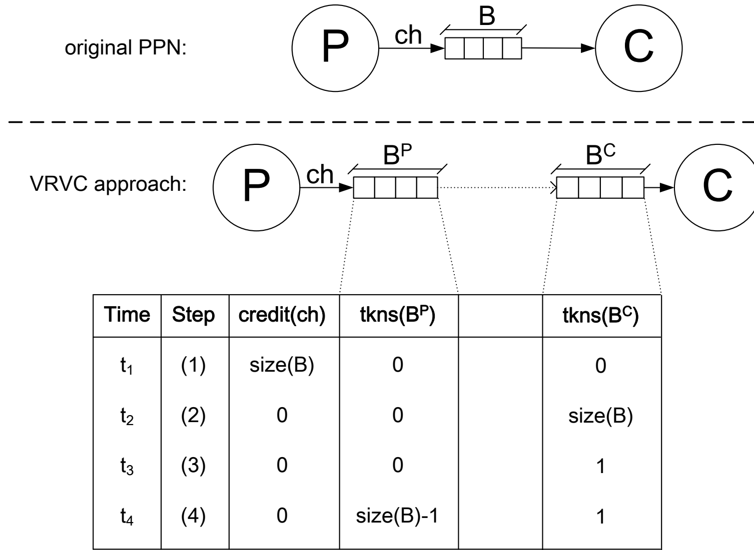
- **Step (1).** This step represents system startup. The credit variable of channel ch is initialized to $size(B)$. The number of tokens stored in B^P and B^C are both zero.
- **Step (2).** After a series of $size(B)$ tokens sent by P, and not consumed by C (recall that, in PPNs, control is completely distributed, so this scenario may occur), the credit variable for channel ch is zero and $tkns(B^C) = size(B)$.
- **Step (3).** Consumer C consumes $size(B) - 1$ tokens, such that, at time instant t_3, $tkns(B^C, t_3) = 1$. However, no virtual token is sent back to P because the parameter n of the VRVC approach is set to $size(B)$ and only $size(B) - 1$ tokens have been consumed.

At the end of Step (3) the total number of tokens stored in B^P and B^C is 1, that is, $\big(tkns(B^P, t_3) + tkns(B^C, t_3)\big) = 1$. However, as mentioned earlier, over time the total number of tokens stored over B^P and B^C should be able to grow up to $size(B)$, to avoid deadlocks. Now, if consumer process C does not consume the last token present in B^C, producer process P cannot send tokens to C. Therefore, P must be able to store up to $size(B) - 1$ tokens in B^P, a scenario which is considered in the next step.

- **Step (4).** This step represents the scenario in which P, at time instant $t_4 > t_3$, has stored $size(B) - 1$ tokens in B^P, and C has only one token left in B^P. Now, since $\big(tkns(B^P, t_4) + tkns(B^C, t_4)\big) = size(B)$, we are sure that process P cannot cause a deadlock due to lack of space in $tkns(B^P)$ and $tkns(B^C)$. Eventually, C will consume the last token left in B^C and send a virtual token back to P, which will increase the credit variable for the corresponding channel and allow new token transfers over the NoC from P to C.

From the scenario described in Example 3.4.1 it follows that FIFO buffers are needed on both the producer and the consumer side. Note that Example 3.4.1 considers a worst-case scenario in the communication between producer and consumer processes. Therefore, the derived size of the buffers, $(size(B) - 1)$ for B^P and $size(B)$ for B^C, is sufficient for all possible scenarios which may arise at run-time.

The pseudocode of the VRVC communication approach is shown in Fig. 3.6. Compared to the VC approach, the behavior of *process_NI_msgs()*, which is used in



Time	Step	credit(ch)	tkns(B^P)		tkns(B^C)
t_1	(1)	size(B)	0		0
t_2	(2)	0	0		size(B)
t_3	(3)	0	0		1
t_4	(4)	0	size(B)-1		1

Figure 3.5: *Communication sequence of a producer-consumer pair, using the VRVC approach, requiring storing of tokens on both the producer and consumer tile. The table in the lower part of the figure shows, at different steps of the communication sequence, the credit value associated to the considered channel, and the number of tokens stored in the producer FIFO buffer B^P and consumer FIFO buffer B^C.*

both the *read* and *write* primitives, changes with regard to the processing of virtual tokens. The first difference is that whenever a virtual token for channel *i* is received, *process_NI_msgs()* consumes it and increases the credit of channel *i* by the parameter n_i. The second difference is that when a virtual token is received and the corresponding FIFO buffer is not empty, as many available tokens as possible are sent to the consumer tile, until the credit for that channel allows so. The credit variable is decreased, accordingly, by the number of tokens sent to the consumer tile.

In the *read* primitive shown in Fig. 3.6, lines 1-2 implement the blocking read behavior, similarly to the VC approach. However, the rest of the primitive is different. In line 3, a token is read from the FIFO buffer which implements channel *CH1*. Line 4 uses an auxiliary variable, *tkns_cns[CH1]*, which keeps track of the number of tokens consumed from *CH1* since the last virtual token sent back (for the corresponding channel) to the producer tile. This auxiliary variable is initialized to zero at startup and is increased for every token consumed by the process from channel *CH1*. In lines 5-7, when this variable reaches the parameter n_{CH1}, a virtual token is sent back to the producer tile and *tkns_cns[CH1]* is reset to zero.

Similarly to the VC approach, in the *write* primitive of VRVC shown in Fig. 3.6, lines 1-2 implement the blocking write behavior. When the control reaches line 3, we are sure that the corresponding FIFO is not full. Then, in the auxiliary function *store_or_send*, the token is either stored in the FIFO buffer corresponding to *CH3* (if the credit variable associated to *CH3* is zero) or sent over the NoC to the consumer

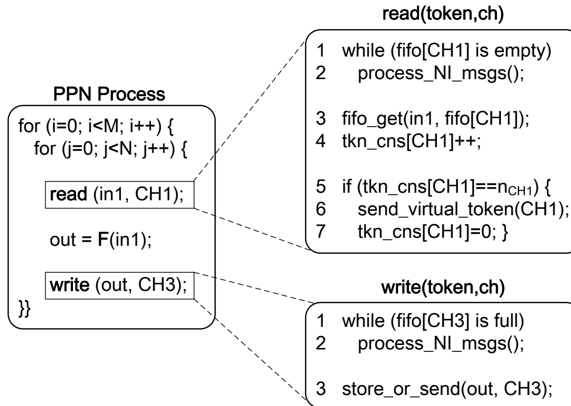


Figure 3.6: *Pseudocode of the VRVC approach. The left part of the figure shows an example of structure of a PPN process. The right side of the figure provides the pseudocodes of read and write PPN primitives as implemented in the VRVC communication approach.*

tile (if the credit variable is greater than zero).

3.4.3 Request-driven approach (R)

This method is very similar to the approach used in [NMSD09] for realizing communication among KPN processes on the Cell BE platform [KDH$^+$05]. In the request-driven approach, the transfer of tokens from the producer tile to the consumer tile is *initiated by the consumer*. This means that every time the consumer is blocked on a read at a given FIFO channel, it sends a *request* to the producer to send new tokens for that channel. The producer, after receiving this request, sends *as many tokens* as it has in its software FIFO implementing that channel.

Since also in this case we need to store tokens both on the producer side and on the consumer side, we need software FIFO structures on both sides. The size of these buffers is set, for each channel i, to match the size of the buffer in the original PPN graph (B_i), therefore $\forall i \in \{1, \cdots, |\mathcal{C}|\}$ $size(B_i^P) = size(B_i^C) = size(B_i)$. This condition guarantees deadlock-free execution on the NoC because: (i) the FIFO buffer residing on the producer tile (B^P) is large enough to avoid deadlocks caused by block on write on the producer side; (ii) after a request sent by the consumer process, B^C is large enough to store the maximum amount of tokens stored in B^P. The structure of a producer-consumer pair using the R approach is shown in Fig. 3.4, case (b). Since the consumer buffer of a channel is empty when a request is made, and given that the FIFO buffers for that channel have the same size on both sides, there is always enough space to store tokens sent by the producer as a consequence of the request.

Fig. 3.7 shows the pseudocode of this PPN communication approach. Similarly to the other communication approaches, it makes use of the auxiliary function *process_NI_msgs()* to process incoming messages of tokens or requests. In the R approach, this function is in charge of reacting to a received request message for a channel

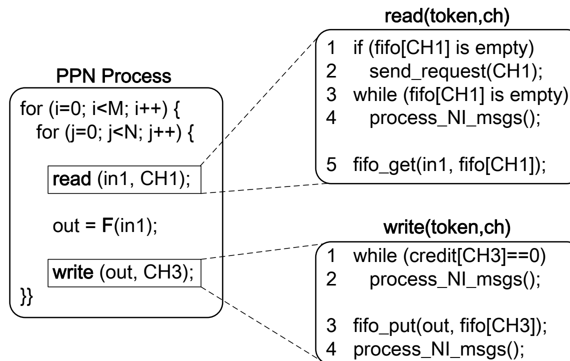


Figure 3.7: *Pseudocode of the Request-driven (R) approach. The left part of the figure shows an example of structure of a PPN process. The right side of the figure provides the pseudocodes of read and write PPN primitives as implemented in the request-driven communication approach.*

with the immediate sending of all the tokens contained in the software FIFO that implements that specific channel.

The *blocking on read* behavior is implemented in lines 1-4 of the read primitive in Fig. 3.7. When the software FIFO of the calling channel is empty, a request is sent to the producer tile of that channel, and the processor keeps executing *process_NI_msgs()* until a message of tokens for the calling channel arrives. The *blocking on write* is implemented in lines 1-2 of the write primitive. When the FIFO of the calling channel (in the example, *CH3*) is full, the processor keeps executing *process_NI_msgs()* until a request for that channel arrives. Line 4 of the write primitive allows a faster response to requests for tokens from consumer processes. In fact, if line 2 of the write primitive is not executed, i.e, if the calling channel is not full, *process_NI_msgs()* is anyway executed in line 4, leading to a faster response to token requests.

3.5 Case Studies

We evaluate the three PPN communication approaches presented in Section 3.4 on two applications, specified as PPNs, with extremely different communication/computation characteristics. The reason is that we want to compare the overhead of the PPN communication approaches between two extremes. The application described in Section 3.5.1 represents the first extreme, when the computation/communication ratio is low and the PPN topology is complicated. The case study described in Section 3.5.2, on the other extreme, is computation dominant and with relatively simple PPN topology. We describe briefly the two case studies in order to get a better understanding of the obtained results. In Section 3.5.3 we also provide an overview of the platform that we use to run the experiments.

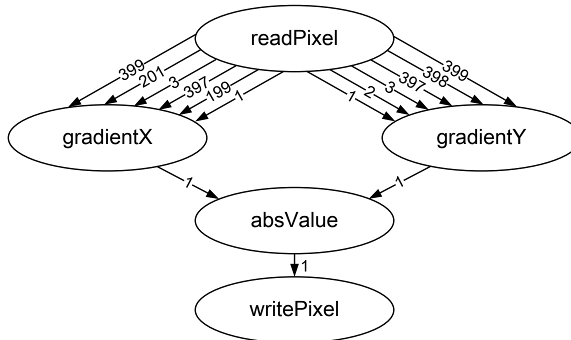


Figure 3.8: *PPN specification of the Sobel filter.*

Table 3.1: *Execution times (in clock cycles) of Sobel functions*

Process	Execution time (c.c.)
readPixel	5
gradientX	31
gradientY	31
absValue	118
writePixel	5

3.5.1 Sobel filter

The Sobel application is an edge-detection algorithm for digital images. Its PPN graph is shown in Fig. 3.8, where the number written over each edge indicate the minimal buffer sizes (expressed in data tokens) needed for that channel, in order to process a 200x122 pixel input image without deadlocks. The PPN processes of this application are very lightweight in terms of computation. The numbers of clock cycles (c.c.) required for one execution of each function are summarized in Table 3.1. The most computationally intensive process is *absValue*, which sums the absolute values of the outputs of the *gradientX* and the *gradientY* processes and normalizes the result. For all of the channels in the graph, the size of exchanged tokens is 4 bytes, and the number of written tokens is 23760. From these metrics it is clear that the Sobel application is largely communication-dominant. Therefore, even before running the actual experiments, we expect this application to perform poorly on NoC-based hardware platforms, where communication is more costly than on platforms with dedicated, point-to-point interconnections. However, we use this example to represent the class of applications in which the communication dominates the computation.

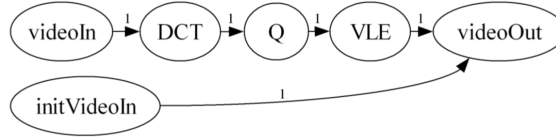


Figure 3.9: *PPN specification of the M-JPEG encoder.*

Table 3.2: *Execution times (in clock cycles) of M-JPEG functions*

Process	Execution time (c.c.)
initVideoIn	18
videoIn	1910
DCT	126386
Q	69238 (avg)
VLE	46688 (avg)
videoOut	1292 (avg)

3.5.2 M-JPEG encoder

The PPN specification of this application is shown in Fig. 3.9. The size of tokens, corresponding to different channels, ranges between 16 and 1024 bytes. All of the channels are written 128 times, except the output of *initVideoIn* which is written only once. The numbers of clock cycles required for the execution of each function of the M-JPEG application are summarized in Table 3.2. This application shows a much simpler communication and synchronization pattern compared to Sobel, and it also has a much higher computation/communication ratio.

3.5.3 Platform setup

The system on which we evaluate our PPN communication approaches is based on a 2x2 mesh of tiles, connected via a custom-built Network-on-Chip. We choose this kind of NoC because, as mentioned in Section 1.1.2, it is the most common and widely studied topology of NoC. However, note that our proposed PPN communication approaches do not depend on the topology of the NoC. Each tile is composed of a MicroBlaze processor, with its program and data memories, and a Network Interface. The platform does not support remote memory access. The system runs at the frequency of 100 MHz.

Each processor has multi-tasking capabilities thanks to the use of the *Xilkernel* operating system, a lightweight, customizable kernel provided by Xilinx. In case of *many-to-one* mapping, i.e. when more than one process are mapped on the same processor, the scheduling is data-driven. This means that a process keeps executing successive iterations until it blocks in reading or writing (recall the PPN process structure of Figure 2.3(b) on page 29). When the process blocks, it yields the processor control to the next process in the ready queue, using round-robin.

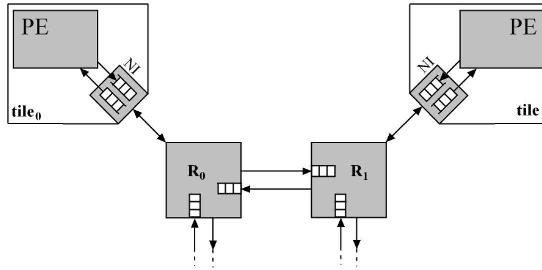


Figure 3.10: *(Part of) the NoC platform structure. The full structure of the adopted NoC structure is a 2x2 mesh of tiles.*

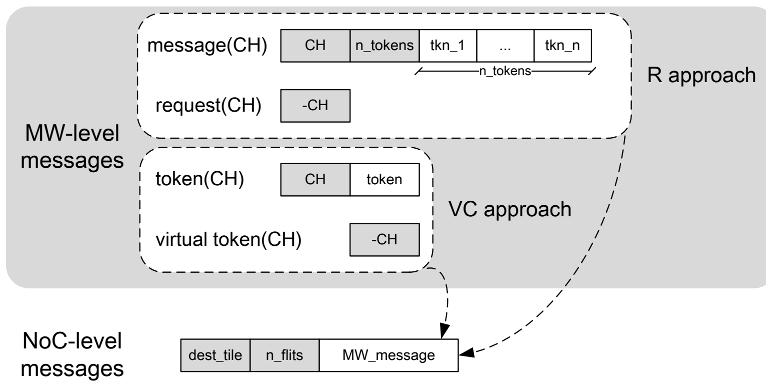


Figure 3.11: *Structure of middleware- and network- level messages.*

As shown in Fig. 3.10, the Network Interface contains only two hardware FIFOs, one for messages which are incoming from the NoC, and one for messages that have to be injected in the NoC. The processor is able to quickly access the status of the incoming hardware FIFO, via a dedicated signal, to see if there are messages to be forwarded from the NI buffer to the software FIFO buffers that implement the channels of the PPN graph. In the opposite direction, when a message has to be sent over the NoC, the processor forwards data from its data memory to the outgoing NI hardware FIFO, then the NI injects the message in the network, with the appropriate header (destination tile and payload size fields). The messages are sent over the NoC using *wormhole switching* (as in [BB04]). As shown in Fig. 3.10, routers (R_0 and R_1) use input buffering to store incoming *flits* (flow control digits, which represent the granules into which messages are split). Moreover, in our implementation the routers use a simple round-robin arbitration policy.

The actual structure of the different kind of messages that are sent over the NoC is represented in Fig. 3.11, for the *VC* and *R* approaches. At NoC-level, the message comprises a NoC header, that indicates the destination tile and the size of the payload, and the payload itself, which we refer to as the middleware (MW)-level message. The

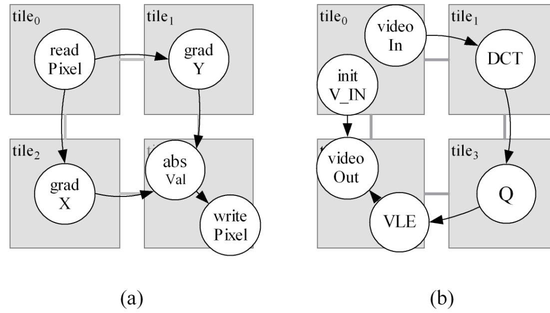


Figure 3.12: *Fixed mappings for Sobel (a) and M-JPEG (b) to test the different PPN communication approaches.*

structure of MW-level messages depends on the PPN communication approach. In *R*, a request for channel number i is implemented as a single flit, with value $-i$. A message used for transferring tokens, instead, has a header composed of two flits (channel number, number of sent tokens) and a payload with the sent tokens. The field that indicates the number of sent tokens (*n_tokens*) is necessary because this number is determined at run-time, when a request for that channel is received. The structure of MW-level messages in *VC* is very similar, the only difference being that there is no need for a *n_tokens* field because in this method sending several tokens in one message is not allowed, i.e. *n_tokens* is always equal to one.

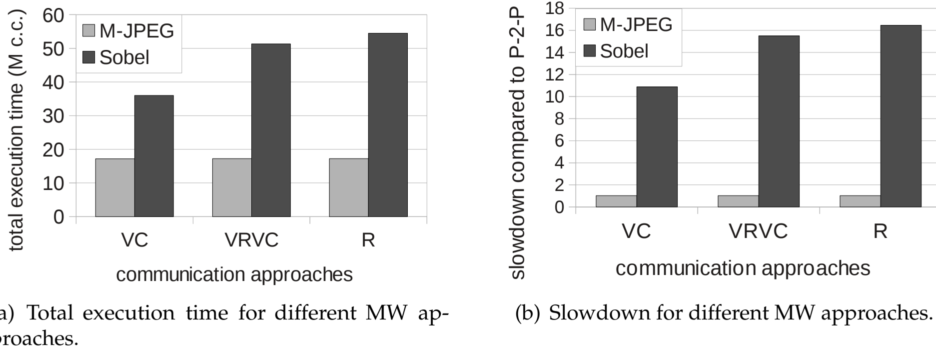
3.6 Experimental Results

The platform described in Section 3.5.3 has been implemented on a Xilinx Virtex-5 FPGA prototyping board. We run the two application case studies of Sections 3.5.1 and 3.5.2, with all the PPN communication approaches proposed in Section 3.4, and obtain the results described below.

3.6.1 Inter-tile communication efficiency

In order to compare the efficiency of inter-tile communication of the different PPN communication approaches, we execute the two case study applications with the fixed mappings shown in Fig. 3.12. We chose these mappings because they expose the maximum amount of inter-tile communication, therefore the obtained results are largely dependent on the efficiency of the considered PPN communication approach.

We found, experimentally, that the parameter n_i of the VRVC approach gives the best performance when set to its maximum value, i.e. when $\forall i \in \{1, \cdots, N_{ch}\}\ n_i = B_i^C$. The performance results, summarized in Fig. 3.13(a), show a large difference of execution time for the Sobel application when using different PPN communication approaches. However, in the M-JPEG case all of the communication approaches yield similar results, due to the much higher computation over communication ratio of



(a) Total execution time for different MW approaches.

(b) Slowdown for different MW approaches.

Figure 3.13: *Sub-figure (a) shows the total execution time of the M-JPEG and Sobel applications obtained with different MW approaches. Notice the large difference of execution time for the Sobel application depending on the used MW approach. By contrast, the execution time for M-JPEG is not affected by the choice of MW approach. Sub-figure (b) compares the performance obtained using our NoC-based platform, together with the proposed MW approaches, with the performance of a customized systems based on point-to-point connections. Notice the large slowdown of the NoC-based implementation for the communication-dominant application, Sobel.*

that application. The VC approach performs much better, compared to the others, in the Sobel application, because its implementation does not require storing of tokens on the producer tile. This leads to a faster communication process, because it avoids the double copy (output variable $\rightarrow$ software FIFO $\rightarrow$ NI buffer) that is necessary in the other cases. We argue that the obtained results may change for NoC platforms with Direct Memory Access (DMA) cores, that can benefit more from sending several tokens with one message, as allowed in the VRVC and R approaches.

In order to evaluate the overhead occurred by using the NoC interconnection and our PPN communication approaches, we implemented customized point-to-point systems, for both Sobel and M-JPEG applications, as a baseline reference. In point-to-point systems, generated using the ESPAM tool [NSD08], a dedicated hardware FIFO is instantiated for each channel of the PPN graph. In this way, the hardware platform perfectly matches the PPN MoC semantics. Obviously, customized point-to-point implementations do not allow for system adaptivity, because all the design decisions (e.g.: process mapping) have to be made at design time. It is clear that in our NoC system we sacrifice performance (especially for communication intensive applications) for adaptivity, the ability of managing the system at run-time, and generality, since the system is able to execute any kind of PPN application. The performance slowdown, when comparing the NoC-based systems with the point-to-point systems is shown in Fig. 3.13(b). It is noticeable that while the Sobel application is highly penalized in the execution on our NoC system, the M-JPEG application performs well because of its higher computation/communication ratio and its regular communication pattern. The reason why the PPN communication on the NoC platform is less efficient, compared to customized point-to-point systems, is mainly twofold. The first reason is that in

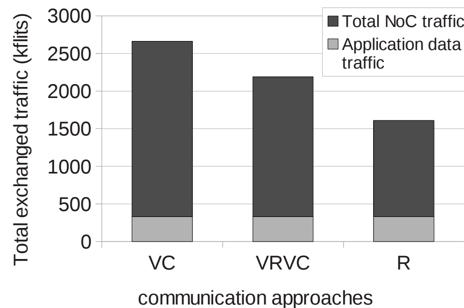


Figure 3.14: *Traffic injected into the NoC by executing Sobel with different MW approaches.*

communicating on the NoC, several PPN channels have to share the same physical channel (the NoC link). The second reason is that in the NoC case we have to use software FIFOs on the producer and on the consumer side, which require additional memory copy operations which would be unnecessary in the case of adoption of hardware FIFOs.

Another important metric when executing applications on a NoC is the amount of generated control traffic overhead. In the *VC* case, for instance, this overhead is represented by the NoC-level and MW-level headers, together with all the traffic generated by the virtual tokens. Ideally, a PPN communication should be designed to generate as less control traffic overhead as possible.

Focusing on the Sobel application, since it has the most complex communication pattern, we profiled the amount of traffic injected in the network, depending on the PPN communication approach that is used. The results, depicted in Fig. 3.14, show that the amount of traffic injected in the NoC (that is, including message headers, messages sent over virtual channels, or requests messages) is much larger than the actual application data traffic. This is because the size of tokens in the Sobel application is extremely small, i.e., only 4 bytes. Note also that the *VC* approach injects much more total traffic into the NoC compared to the *R* approach. This large difference can be explained by two factors. The first factor is the overhead of message headers. On the one hand, in the *VC* method, each token travels in the NoC with its own header. On the other hand, in the *R* case, the producer sends as many token as present in its software FIFO, in the same message and therefore with the same header. The second factor is that the traffic on virtual channels in *VC* is much more than the traffic generated by requests in *R*. This is because in the *VC* approach a virtual token is sent back to the producer for every consumed token, while in the *R* approach the requests are made less frequently, just when the consumer is blocked on reading.

3.6.2 System adaptivity support

All the proposed PPN communication approaches (*VC*, *VRVC*, *R*) allow to change at run-time the mapping of PPN processes to tiles of the system. Process remapping is

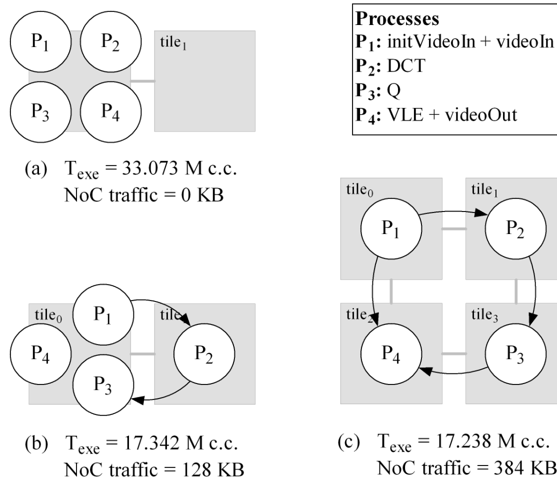


Figure 3.15: *Execution time and generated traffic as a function of the process mapping. Only inter-tile communication links are depicted.*

allowed because all the PPN communication approaches exploit a *middleware table*, which keeps track of the source and destination tiles of each channel of the PPN. We recall that these tables are used to convert the generic PPN primitives (see for instance *READ, WRITE* in Figure 2.3(b) on page 29) to the corresponding hardware platform communication primitives, where the source tile and destination tile of a channel are precisely specified. When a remapping of processes is performed, the middleware tables are changed accordingly.

Note that the actual mechanism used to perform run-time remapping of processes is proposed and explained in the next chapter, Chapter 4. However, the approaches presented in this chapter ensure that, once the remapping procedure is completed, PPN processes can communicate correctly also in the new spatial mapping.

The possibility of choosing a different mapping at run-time can be exploited by run-time management algorithms, or simply by the user of the system, to trade between performance and other metrics, such as power. For instance, in Fig. 3.15 we show the performance results obtained with different mappings of the M-JPEG application. For each mapping, Fig. 3.15 shows the total execution time (T_{exe}) and total exchanged traffic over the NoC (*NoC traffic*). Mapping (a) in the figure has only one active tile, whereas mapping (b) requires two active tiles. Therefore, we can infer that mapping (a) is more power efficient than mapping (b), also because the former uses no power to communicate over the NoC. However, T_{exe} achieved by mapping (b) is almost half of the one of mapping (a), thus mapping (b) is preferable when the system has to provide high performance. Finally, by comparing mapping (b) and (c) in Fig. 3.15, we see that exploring more than two tiles when mapping the M-JPEG application has only marginal performance benefits.

3.7 Discussion

From the experimental results reported in Section 3.6.1 we can quantitatively compare the three PPN communication approaches proposed and evaluated in this chapter. For each communication approach, we summarize the advantages (+) and disadvantages (−) in the following list.

- Virtual Connector (*VC*):
  - + Outperforms all the other approaches (*VRVC*, *R*) for applications with low computation over communication ratio;
  - − Requires a credit-based system, with frequent synchronization between producer and consumer processes.
- Virtual Connector with Variable Rate (*VRVC*):
  - + Achieves slightly higher performance than the *R* approach for applications with low computation over communication ratio;
  - − Requires a credit-based system, and synchronization between producer and consumer processes with a frequency which depends on a parameter that can be tuned.
- Request-driven (*R*):
  - + Simpler implementation compared to the other approaches, without a credit-based system and with less synchronization points;
  - + Achieves performance nearly identical to *VC* and *VRVC* when the computation over communication ratio is high;
  - − Achieves low performance for applications in which the computation over communication ratio is low.

From the above comparison, it follows that if a designer needs to map a PPN application with low computation over communication ratio on a NoC based execution platform and *with a static mapping*, the *VC* approach is preferable.

However, when run-time remapping of processes is necessary, the *VC* approach is less appealing because it requires frequent synchronization between producer and consumer processes. This is especially true when process remapping is needed to achieve fault tolerance, with process migrations that can be triggered at any time by hardware faults. Note that with our proposed middleware we want to address also this kind of scenario. Therefore, in Chapter 4, we propose a process migration mechanism which is based on the *R* approach. In fact, as mentioned above, *R* has two main advantages. First, it has less synchronization points between producer and consumer processes, and it is easy to implement. Second, it achieves identical performance for applications with high computation over communication ratio, the kind of applications which are more likely to be executed on NoC platforms.