Middleware Approaches for Adaptivity of Kahn Process Networks on Networks-on-Chip

Emanuele Cannella*, Onur Derin†, Todor Stefanov*
*LIACS, Leiden University
Leiden, The Netherlands
†ALaRI, University of Lugano,
Lugano, Switzerland
Email: cannella@liacs.nl*, derino@alari.ch†, stefanov@liacs.nl*

***Abstract*—We investigate and propose a number of different middleware approaches, namely *virtual connector*, *virtual connector with variable rate*, and *request-driven*, which implement the semantics of Kahn Process Networks on Network-on-Chip architectures. All of the presented solutions allow for run-time system adaptivity. We implement the approaches on a Network-on-Chip multiprocessor platform prototyped on an FPGA. Their comparison in terms of the introduced overhead is presented on two case studies with different communication characteristics. We found out that the virtual connector mechanism outperforms other approaches in the communication-intensive application. In the other case study, which has a higher computation/communication ratio, the middleware approaches show similar performance.**

***Index Terms*—System adaptivity; Kahn Process Networks; Middleware; Networks-on-Chip;**

I. Introduction

The complexity of multiprocessor systems on chip (MPSoCs) is rapidly increasing, driven by the technology improvement and the adoption of more and more complex applications in consumer electronics. Programming such complex systems at a low level of abstraction is extremely difficult and error-prone. A promising way to raise the level of abstraction is using models of computation (MoCs) to specify applications. Among these MoCs, Kahn Process Networks (KPNs) have been widely studied and used for streaming/multimedia applications.

KPNs are composed by concurrent and autonomous processes that communicate between each other using unbounded FIFO channels. However, when implementing such a model of computation on real platforms, FIFO channels size must be bounded. In our approach, we use the `pn` compiler [1] to automatically convert static affine nested loop programs (SANLPs) to parallel KPN specifications and to determine the buffer sizes that guarantee deadlock-free execution. Thus, using the KPN model of computation allows us to program an MPSoC in a systematic and automated way. Furthermore, in KPNs, the control is completely distributed, as well as the memories. This represents a good match with the emerging MPSoC architectures, in which processing elements and memories are usually distributed.

Another favorable feature of the KPN MoC is that its simple operational semantics allows for the easy adoption of system adaptivity mechanisms. System adaptivity is becoming increasingly important in the MPSoC domain for several reasons, such as dynamic variation of quality of service requirements, fault tolerance, or power efficiency.

Networks-on-Chip (NoCs) [2] are emerging communication infrastructures for MPSoCs that, among many other advantages, allow for system adaptivity. This is because the same NoC platform can be used to run different applications, or to run the same application with different mapping of processes. However, there is a mismatch between the generic structure of the NoCs and the semantics of the KPN MoC. Therefore, in this paper we investigate and propose several approaches to overcome this mismatch. All of the proposed approaches consider system adaptivity as a driving objective and do not require specific hardware support from the platform to realize the inter-tile communication between processes.

The remainder of the paper is organized as follows: Section I-A continues the introduction by stating the problem along with a list of related work in Section I-C. The proposed and investigated middleware approaches are described in detail in Section II. The applications used in the middleware approaches evaluation are explained in Section III followed by the performance results in Section IV. Finally, Section V concludes the paper.

A. Problem statement

The main problem addressed in our work is the efficient implementation of a middleware allowing the execution of applications modeled as KPNs on Network-on-Chip platforms. The first requirement is that this middleware must respect the KPN semantics. That is, processes must *block on read*, when trying to get a data token from an empty FIFO, and *block on write*, when trying to write data to a full FIFO. Moreover, we want our middleware to be application-independent and oriented to system adaptivity.

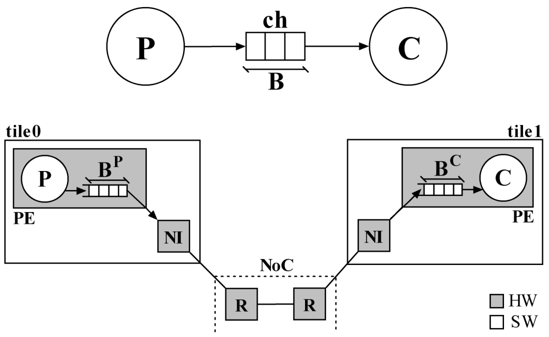
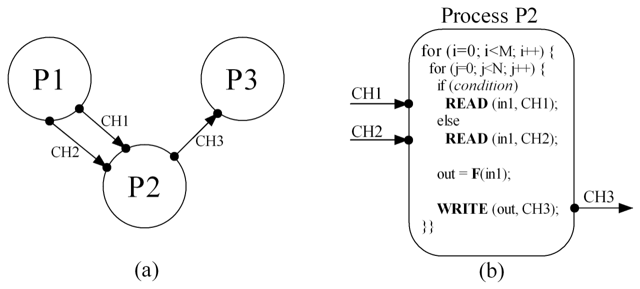


Fig. 1. Producer-consumer pair with FIFO buffer split over two tiles.



```
for (i=0; i<M; i++) {
  for (j=0; j<N; j++) {
    if (condition)
      READ (in1, CH1);
    else
      READ (in1, CH2);

    out = F(in1);

    WRITE (out, CH3);
}}
```

Fig. 2. Example of a KPN (a) and structure of process $P2$ (b).

The communication and synchronization problem when mapping KPNs on a NoC is depicted in Fig. 1. Consider a producer P and a consumer C connected through an asynchronous communication FIFO link of size B. If both the producer and the consumer can directly access the status register of this FIFO buffer, to check if it is empty or full, implementing the KPN semantics is straightforward. However, in NoC implementations with no direct remote memory access, processes can exchange tokens only via the network. Thus, we have to split the buffer B in B^P and B^C, one on the producer tile and one on the consumer tile. We want to implement the KPN semantics without a dedicated support from the underlying architecture that allows checking for the status of the remote queues. If B is the minimum buffer size that guarantees deadlock-free execution of the original KPN graph, the size of B^P and B^C must be set such that $B^P + B^C \geq B$.

We do not require support for multiple hardware FIFOs on each NoC tile. The only hardware buffer of a tile resides in the Network Interface (NI). We just rely on the ability to transfer tokens, in both directions, from this buffer to the *software FIFOs* which implement the channels of our KPN.

Even if the consumer can only 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. There are several ways to notify the producer about the status of the buffer on the consumer side, and we will compare the approaches that we have investigated in Section II.

Furthermore, we want our middleware to take care of the distribution of processes among the NoC tiles with no influence on the application designer. This means that we want to maintain the code structure of the KPN application processes, an example of which is shown in Fig. 2(b). In particular, we want the communication primitives of KPN processes (*read*, *write*) to remain generic, without the notion of process mapping or platform details. These generic primitives are then translated by the middleware in mapping- and platform-dependent API function calls.

B. Paper contributions

The contributions of our paper are two-fold. On the one hand, we propose different middleware approaches that enable mapping-independent and efficient execution of KPN applications on NoC-based platforms. On the other hand, it enables the run-time remapping capability of processes among the tiles of the NoC, thus enabling their run-time adaptable execution.

C. Related work

Kahn process networks (KPNs) [3] are a widely studied distributed model of computation used for describing systems where streams of data are transformed by processes executing in sequence or parallel. Previous research on the use of KPNs in multiprocessor embedded devices has been mainly focusing on the design of frameworks which employ them as a model for application specification [4], [5], [6], and which aim at supporting and optimizing the mapping of KPN processes on the nodes of a reference platform [7], [8]. In [4], [5], different methods and tools are proposed for automatically generating KPN application models from 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. A design phase usually follows in which software synthesis for multi-processor systems [6], [8], or architecture synthesis for FPGA platforms [4] is implemented.

The approaches described above, which map applications described as KPNs to customized platforms, have a strong coupling between the application and the platform. Running a different application on the generated platform would not be possible or, even if possible, would give bad performance results. We adopt a different approach where we start by the assumption that we have a platform equipped with heterogeneous cores well interconnected with a NoC. We provide a KPN API for this platform that the KPN application processes will comply to. Most importantly, the application code remains the same in all possible mappings of the processes. This is achieved by the proposed intermediate layer, called *middleware*, that includes the mapping related information and implements the KPN API.

This approach, where software synthesis relies on the high level APIs provided by the reference platform for facilitating

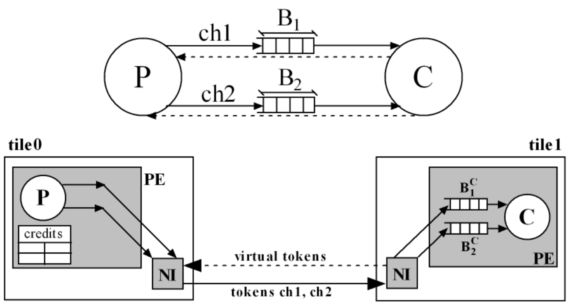


Fig. 3. Producer-consumer pair using the virtual connector method.

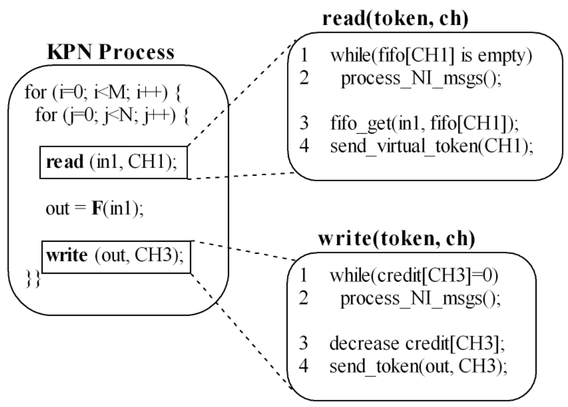
the programming of a multiprocessor system, can be seen elsewhere. The trend from single core design to many core design has forced to consider inter-processor communication issues for passing the data between the cores. One of the emerged message passing communication API is Multicore Association's Communication API (MCAPI) [9] 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 [10]. However these MPI standards are not quite fit for the KPN semantics [11] and building the semantics on top of their primitives is an additional overhead that may not be afforded.

The communication and synchronization problem when implementing KPNs over multi-processor platforms without hardware support for FIFO buffers has been considered in [12] and [8]. In [12] the *receiver-initiated* method has been proposed and evaluated for the Cell BE platform. On the same hardware platform, [8] proposes a different protocol, which makes use of mailboxes and *windowed FIFOs*. The difference with our work presented in this paper 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 [11] 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 paper we propose yet another approach, namely *virtual connector with variable rate*.

In [13] the problem of implementing the KPN semantics on a NoC is addressed. 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 our work system adaptivity is considered because the middleware enables run-time management and the platform is generic, i.e. it allows the execution of any application specified as a KPN.

An approach to guarantee blocking write behavior is also used in [14]. That work proposes the use of dedicated operating system communication primitives, which guarantee that the remote FIFO buffer is not full before sending messages through a simple request/acknowledge protocol. Compared to this kind of protocol, the middleware approaches described in our paper assume a more proactive behavior of the consumer processes to guarantee the blocking on write.



```
for (i=0; i<M; i++) {
  for (j=0; j<N; j++) {
    read (in1, CH1);
    out = F(in1);
    write (out, CH3);
}}
```

```
while(fifo[CH1] is empty)
  process_NI_msgs();

fifo_get(in1, fifo[CH1]);
send_virtual_token(CH1);
```

```
while(credit[CH3]=0)
  process_NI_msgs();

decrease credit[CH3];
send_token(out, CH3);
```

Fig. 4. Pseudocode of the VC approach.

II. Middleware approaches

This section describes the different solutions that we have explored for the implementation of KPN process communication and synchronization on a tiled NoC-based architecture. Basically, the proposed approaches differ in the frequency of acknowledgment messages sent from the consumer process to the producer process about the status of the consumer FIFO buffers.

In all of the approaches described below, system adaptivity is taken into account by using dedicated middleware tables that list, among other information, the source and destination tile for each channel of the KPN graph. For instance, when the middleware is up to send a packet to the consumer of a specific channel, it will check in the table what is the current destination of that channel. Then, it will place the packet in the NI output buffer, with the appropriate destination field of the header. These middleware tables can be updated at run-time to allow changing dynamically the mapping of application processes over the tiles.

A. Virtual connector approach (VC)

In the virtual connector approach, which is depicted in Fig. 3, for every channel in the original KPN graph we add a virtual 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 [11], to the needs of our system implementation. In that work the proposed middleware is *active*, meaning that it is implemented using separate threads which deal with the KPN communication, while in our implementation the middleware is *static*, with no separate threads for communication. Although a comparison of the static and active implementations may be worthwhile to do, for the moment we adopt the static approach with the argument that the scheduling and synchronization of additional middleware processes may introduce an additional overhead due to the context switching times.

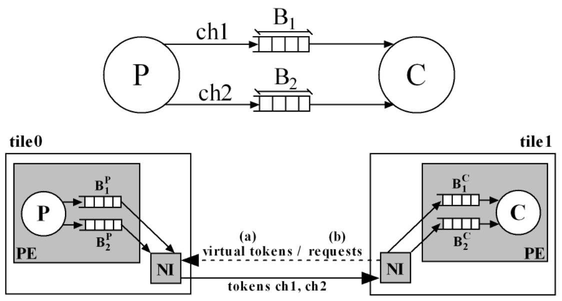
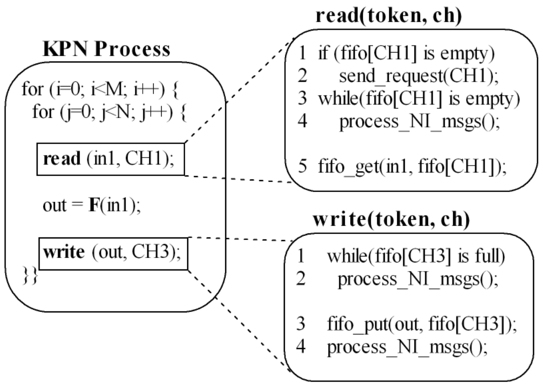


Fig. 5. Producer-consumer implementation: when using the VRVC, the producer receives back virtual tokens (a); when using R, it receives requests (b).



```
KPN Process
for (i=0; i<M; i++) {
  for (j=0; j<N; j++) {
    read (in1, CH1);
    out = F(in1);
    write (out, CH3);
}}
```

```
read(token, ch)
1 if (fifo[CH1] is empty)
2     send_request(CH1);
3 while(fifo[CH1] is empty)
4     process_NI_msgs();

5 fifo_get(in1, fifo[CH1]);
```

```
write(token, ch)
1  while(fifo[CH3] is full)
2    process_NI_msgs();

3  fifo_put(out, fifo[CH3]);
4  process_NI_msgs();
```

Fig. 6. Pseudocode of the R approach.

For each channel in the original KPN 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 KPN 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. This is due to the fact that the credit-system guarantees that enough locations are free on the remote buffers before sending a token. Therefore, referring back to Fig. 1, in this approach for each channel i, $B_i^C = B_i$ and $B_i^P = 0$.

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 ($credit_i = B_i^C$), because all of its slots are free. 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.

The pseudocode of the VC approach is described in Fig. 4. Both the *read* and *write* primitives use an auxiliary function, *process_NI_msgs()*, that is used when blocking on read or on write. This function checks the status of the NI buffer for incoming packets. If the buffer is not empty, it processes one packet at a time, until all the incoming packets are consumed, in the following way. If the packet is an incoming token for channel *i*, it stores the token in the software FIFO which implements channel *i*. If it is a virtual token for channel *j*, it consumes the packet and increase the credit of channel *j*.

Lines 1-2 of the *read* primitive 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 of *CH1*. This is actually done by putting in the NI outgoing buffer a packet representing a virtual token for channel *CH1*, as shown in Fig. 10.

Similarly, in the *write* primitive, 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 actually done by putting in the NI outgoing buffer a packet representing the token (refer again to Fig. 10).

B. Virtual connector with variable rate approach (VRVC)

This approach represents a variant of the *virtual connector* described above. The basic idea is that instead of sending one virtual token to the producer for *every* consumed token of channel i, the consumer sends it after n_i consumed tokens, where n_i is a parameter that can be set such that $\forall i \in \{1, \cdots, N_{ch}\}$ $1 \leq n_i \leq B_i$, where N_{ch} represents the number of channels in the KPN 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 packets.

Since the sending back of virtual tokens does not happen for every consumed token, in some cases the KPN graph properties require to store, also at the producer side, tokens for the channels in order to avoid deadlocks. This requires the adoption of software FIFO buffers also on the producer side. In the most generic case, the size of these buffers should be as large as the original buffer in the KPN graph. This means that $\forall i \in \{1, \cdots, N_{ch}\}$ $B_i^P = B_i^C = B_i$, as depicted in Fig. 5, case (a). The pseudocode for the VRVC method is omitted for the sake of brevity.

C. Request-driven approach (R)

This method is very similar to the approach used in [12] for realizing the FIFO communication on the Cell BE platform.

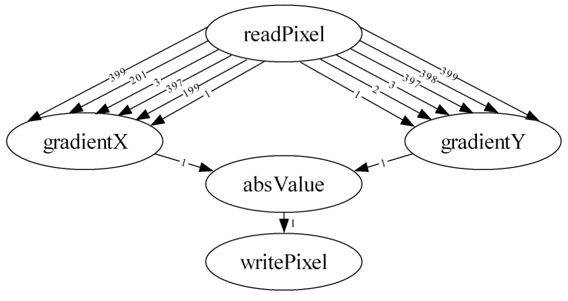


Fig. 7. KPN specification of the Sobel filter.

TABLE I
EXECUTION TIMES OF SOBEL FUNCTIONS

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

In this 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 queue in the original KPN graph (B_i), such that $\forall i \in \{1, \cdots, N_{ch}\}$ $B_i^P = B_i^C = B_i$. This condition guarantees deadlock-free execution on the NoC and it is the same as in the *VRVC* approach. The structure of a producer-consumer pair using the *R* approach is shown in Fig. 5, 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. 6 shows the pseudocode of this middleware approach. Similarly to the VC middleware, it makes use of the auxiliary function *process_NI_msgs()* to process incoming packets of tokens or requests. The main difference in this case is that this function is in charge of reacting to a received request message for a channel 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. 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 packet 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.

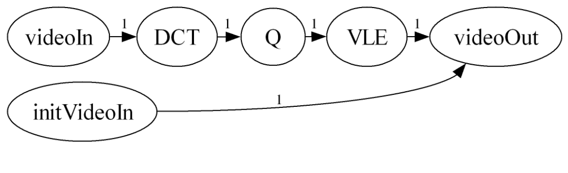


Fig. 8. KPN specification of the M-JPEG encoder.

TABLE II
EXECUTION TIMES OF M-JPEG FUNCTIONS

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

III. CASE STUDIES

We evaluate the three middleware approaches presented in Section II on two KPN applications with extremely different communication/computation characteristics. The reason is that we want to compare the overhead of the middlewares between two extremes. The application described in Section III-A represents the worst case (the first extreme), when the computation/communication ratio is low and the KPN topology is complicated. The case study described in Section III-B, on the other extreme, is computation dominant and with relatively simple KPN topology, therefore represents the best case. We describe briefly the two case studies in order to allow a better understanding of the obtained results. We also provide an overview of the platform that we use to run the experiments.

A. Sobel filter

The Sobel application is an edge-detection algorithm for digital images. Its KPN graph is shown in Fig. 7, where the numbers over the edges indicate the minimal buffer sizes needed for processing a 200x122 pixel input image. The KPN processes which comprise this application are very lightweight in terms of computation. The numbers of clock cycles required for one execution of each function are summarized in Table I. 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.

B. M-JPEG encoder

The KPN specification of this application is shown in Fig. 8. The size of tokens ranges between 16 and 1024 bytes, and all of the channels are written 128 times, except the output of *initVideoIn* which is written only once. The numbers of clock

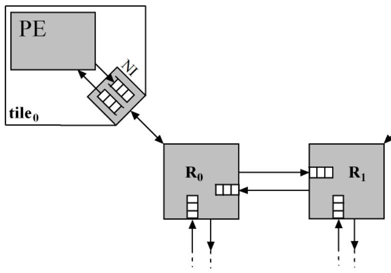


Fig. 9. NoC platform structure.

cycles required for the execution of each function of the M-JPEG application are summarized in Table II. This application shows a much simpler communication and synchronization pattern compared to Sobel, and it also has a much higher computation/communication ratio.

C. Platform setup

The system on which we evaluated our middleware approaches is based on a 2x2 mesh of tiles, connected via a Network-on-Chip. Each tile is composed by 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 runs as long as it blocks in reading or writing. When the process blocks, it yields the processor control to the next process in the ready queue.

As shown in Fig. 9, the Network Interface contains the only two hardware FIFOs inside the tile, one for packets which are incoming from the NoC, and one for packets 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 SW FIFO buffers that implement the channels of the KPN graph. In the opposite direction, when a packet 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 packet in the network, with the appropriate header (destination tile and payload size fields). The packets are sent over the NoC using *wormhole routing*. As shown in Fig. 9, routers (R_0 and R_1) use input buffering to store incoming flits. 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. 10, for the *VC* and *R* approaches. At NoC-level, the packet comprises a NoC header, that indicates the destination tile and the size of the payload, and the payload itself, which is the middleware (MW)-level packet. The structure of MW-level packets depends on the middleware approach. In *R*, a request

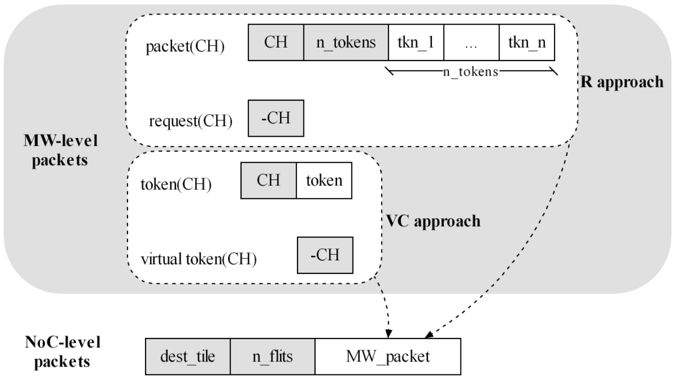


Fig. 10. Structure of middleware- and network- level packets.

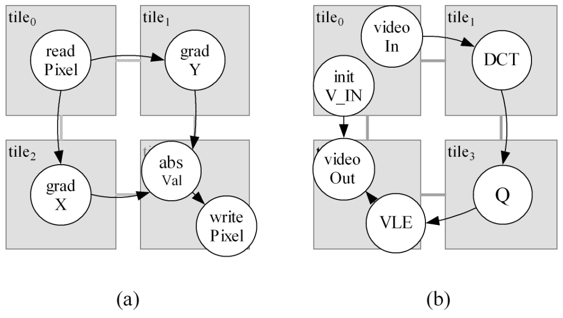


Fig. 11. Fixed mappings for Sobel (a) and M-JPEG (b) to test the different middleware approaches.

for channel number i is implemented as a single flit, with value $-i$. A packet 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 packets in *VC* is very similar, the only difference is that there is no need for a *n_tokens* field because in this method there is no packetization of tokens, i.e. *n_tokens* is always equal to one.

IV. EXPERIMENTAL RESULTS

The platform described in Section III has been implemented on a Virtex5 FPGA prototyping board. We run the two application case studies, with all the middleware approaches proposed in Section II, to obtain the results described below.

A. Inter-tile communication efficiency

In order to compare the efficiency of inter-tile communication of the different middleware approaches, we execute the two case study applications with fixed mappings, which are shown in Fig. 11. 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 middleware.

We found out experimentally that the parameter n_i of the VRVC case gives the best performance when is set to its

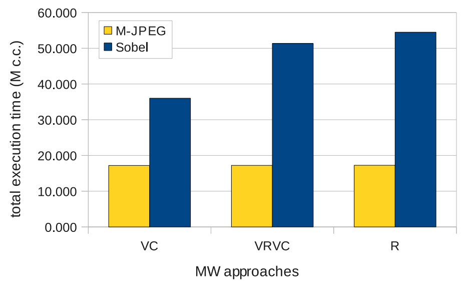


Fig. 12. Total execution time for different MW approaches.

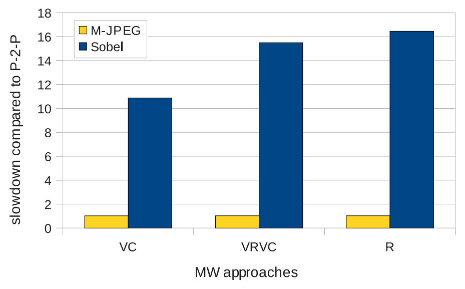


Fig. 13. Slowdown for different MW approaches.

maximum value, i.e. when $\forall i \in \{1, \cdots, N_{ch}\}$ $n_i = B_i^C$. The performance results, summarized in Fig. 12, show a large difference of execution time for the Sobel application when using different middleware approaches. However, in the M-JPEG case all of the middleware approaches yield to similar results. 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 the packetization of tokens allowed in the VRVC and R approaches.

In order to evaluate the overhead imposed by the use of the NoC interconnection and our middleware approaches, we implemented customized point-to-point systems, for both applications, as a baseline reference. In point-to-point systems, generated using the ESPAM tool [4], a dedicated hardware FIFO is instantiated for each channel of the KPN graph. In this way, the hardware platform perfectly matches the KPN 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 KPN application. The performance slowdown, when comparing the NoC system with the point-to-point

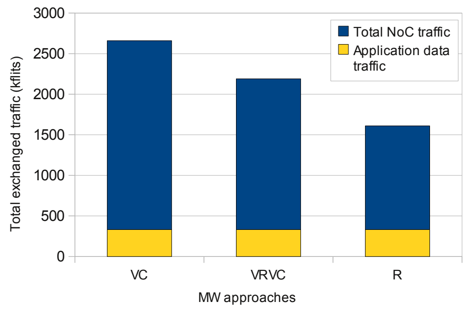


Fig. 14. Traffic injected into the NoC by executing Sobel with different MW approaches.

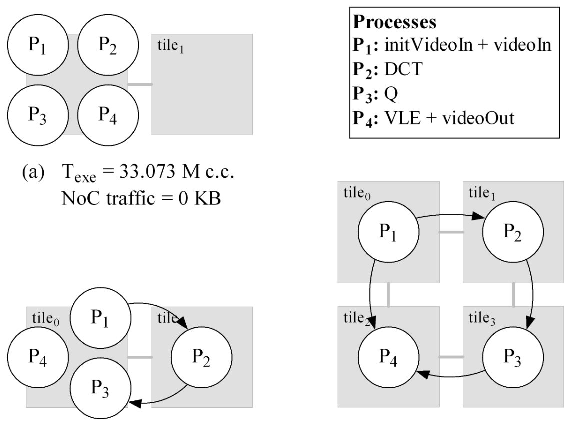
systems is shown in Fig. 13. It is noticeable that while the Sobel application is highly penalized in the execution on our NoC system, the M-JPEG application performs well because of its higher computation/communication ratio and its regular communication pattern. The reasons why the communication onto the NoC platform is less efficient are mainly twofold. The first reason is that in this implementation, several KPN 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 middleware 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 middleware approach that is used. The results, depicted in Fig. 14, show two extremes: the *VC* and *R* methods. This large difference can be explained by two factors. The first factor is the overhead of packet headers. On the one hand, in the *VC* method, since there is no packetization of tokens, 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 packet 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* case a virtual token is sent back to the producer for every consumed token, while in the *R* case the requests are made less frequently, just when the consumer is blocked on reading.

B. System adaptivity support

In order to assess the benefits of a different application process mapping over the NoC, we run the M-JPEG application using the mappings shown in Fig. 15. We proved



(a) T_{exe} = 33.073 M c.c.
NoC traffic = 0 KB

(b) T_{exe} = 17.342 M c.c.
NoC traffic = 128 KB

(c) T_{exe} = 17.238 M c.c.
NoC traffic = 384 KB

Fig. 15. Execution time and generated traffic as a function of the process mapping. Only inter-tile communication links are depicted.

experimentally that different mappings actively impact metrics like total execution time (T_{exe} in Fig. 15) and total exchanged traffic over the NoC (*NoC traffic* in Fig. 15). In the current implementation, the code for all of the processes are mapped on *each* tile of the system, as separate threads (*process replication*). A way to implement the run-time remapping is as follows: threads are activated or stopped on selected cores using the underlying operating system support, and the dedicated middleware tables are changed, with the updated source and destination tiles for each channel of the KPN. For instance, in Fig. 15(a) all the processes of the M-JPEG application are executed on one tile, and the communication between processes does not happen via the NoC. However, in Fig. 15(b) processes P_1, P_3, P_4 are executed on one tile and process P_2 runs on another tile. The middleware tables which describe the source and destination of each channel of the KPN are changed accordingly. Doing this at run-time is our ongoing work. However, our main message is that the middleware approach enables the described run-time remapping feature by exposing the middleware tables and the FIFOs of the application. The implemented middleware is capable of supporting all mappings, thus allowing system adaptivity.

V. Conclusions

We proposed and implemented three middleware approaches to execute Kahn Process Networks on Network-on-Chip architectures. Experimental results on two applications with very different computation and communication characteristics showed that the *virtual connector* approach outperforms the others when implementing communication-dominant applications. However, especially for this kind of applications, the price we pay for system adaptivity and generality is large in terms of performance, compared to customized point-to-point systems. On the contrary, when the computation/communication ratio of an application is higher, as in the second case study, the overhead introduced by the execution on NoC with all the proposed middlewares is much lower.

VI. Acknowledgments

This work was funded by the European Commission under the Project MADNESS (No. FP7-ICT-2009-4-248424) and with a collaboration grant by the HiPEAC Network of Excellence. The paper reflects only the authors' view; the European Commission is not liable for any use that may be made of the information contained herein.

References

[1] S. Verdoolaege, H. Nikolov, and T. Stefanov, "pn: A Tool for Improved Derivation of Process Networks," *EURASIP J. Embedded Syst.*, vol. 2007, pp. 19–19, January 2007.

[2] G. De Micheli and L. Benini, *Networks on Chips: Technology and Tools*. Morgan Kaufmann, 2006.

[3] G. Kahn, "The semantics of a simple language for parallel programming," in *Information Processing '74: Proceedings of the IFIP Congress*, J. L. Rosenfeld, Ed. New York, NY: North-Holland, 1974, pp. 471–475.

[4] H. Nikolov, T. Stefanov, and E. Deprettere, "Systematic and Automated Multiprocessor System Design, Programming, and Implementation," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 27, no. 3, pp. 542–555, 2008.

[5] A. Nieuwland, J. Kang, O. P. Gangwal, R. Sethuraman, N. Bus, K. Goossens, R. Peset Llopis, and P. Lippens, "C-heap: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems," *Design Automation for Embedded Systems*, vol. 7, pp. 233–270, 2002, 10.1023/A:1019782306621.

[6] S. Kwon, Y. Kim, W.-C. Jeun, S. Ha, and Y. Paek, "A retargetable parallel-programming framework for MPSoC," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 13, pp. 39:1–39:18, July 2008.

[7] I. Bacivarov, W. Haid, K. Huang, and L. Thiele, "Methods and Tools for Mapping Process Networks onto Multi-Processor Systems-On-Chip," in *Handbook of Signal Processing Systems*, S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, Eds. Springer, Oct. 2010, pp. 1007—1040.

[8] W. Haid, L. Schor, K. Huang, I. Bacivarov, and L. Thiele, "Efficient Execution of Kahn Process Networks on Multi-Processor Systems Using Protothreads and Windowed FIFOs," in *Proc. IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*. Grenoble, France: IEEE, 2009, pp. 35–44.

[9] "Multicore associations communication api." [Online]. Available: http://www.multicore-association.org

[10] "A high performance message passing library." [Online]. Available: http://www.open-mpi.org/

[11] O. Derin, E. Diken, and L. Fiorin, "A Middleware Approach to Achieving Fault-tolerance of Kahn Process Networks on Networks-on-Chips," *International Journal of Reconfigurable Computing*, vol. 2011, no. Article ID 295385, p. 14 pages, February 2011, selected Papers from the International Workshop on Reconfigurable Communication-centric Systems on Chip (ReCoSoC' 2010).

[12] D. Nadezhkin, S. Meijer, T. Stefanov, and E. Deprettere, "Realizing FIFO Communication when Mapping Kahn Process Networks onto the Cell," in *Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, ser. SAMOS '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 308–317.

[13] A. B. Nejad, K. Goossens, J. Walters, and B. Kienhuis, "Mapping KPN Models of Streaming Applications on A Network-on-Chip Platform," in *ProRISC 2009: Proceedings of the Workshop on Signal Processing, Integrated Systems and Circuits*, November 2009.

[14] Gabriel Marchesan Almeida and Gilles Sassatelli and Pascal Benoit and Nicolas Saint-Jean and Sameer Varyani and Lionel Torres and Michel Robert, "An Adaptive Message Passing MPSoC Framework," *International Journal of Reconfigurable Computing*, vol. 2009, p. 20, 2009.