Adaptivity Support for MPSoCs Based on Process Migration in Polyhedral Process Networks

Cannella, E.; Derin, O.; Meloni, P.; Tuveri, G.; Stefanov, T.P.

Hindawi Publishing Corporation
VLSI Design
Volume 2012, Article ID 987209, 17 pages
doi:10.1155/2012/987209

Research Article

Adaptivity Support for MPSoCs Based on Process Migration in Polyhedral Process Networks

Emanuele Cannella,[1] Onur Derin,[2] Paolo Meloni,[3] Giuseppe Tuveri,[3] and Todor Stefanov[1]

[1] *LIACS, Leiden University, 2333 CA Leiden, The Netherlands*
[2] *ALaRI, Faculty of Informatics, University of Lugano, 6904 Lugano, Switzerland*
[3] *DIEE, Faculty of Engineering, University of Cagliari, 09123 Cagliari, Italy*

Correspondence should be addressed to Emanuele Cannella, cannella@liacs.nl

Received 30 August 2011; Accepted 25 October 2011

Academic Editor: Luigi Raffo

System adaptivity is becoming an important feature of modern embedded multiprocessor systems. To achieve the goal of system adaptivity when executing Polyhedral Process Networks (PPNs) on a generic tiled Network-on-Chip (NoC) MPSoC platform, we propose an approach to enable the run-time migration of processes among the available platform resources. In our approach, process migration is allowed by a middleware layer which comprises two main components. The first component concerns the inter-tile data communication between processes. We develop and evaluate a number of different communication approaches which implement the semantics of the PPN model of computation on a generic NoC platform. The presented communication approaches do not depend on the mapping of processes and have been implemented on a Network-on-Chip multiprocessor platform prototyped on an FPGA. Their comparison in terms of the introduced overhead is presented in two case studies with different communication characteristics. The second middleware component allows the actual run-time migration of PPN processes. To this end, we propose and evaluate a process migration mechanism which leverages the PPN model of computation to guarantee a predictable and efficient migration procedure. The efficiency and applicability of the proposed migration mechanism is shown in a real-life case study.

1. Introduction

The technology improvement and the adoption of more and more complex applications in consumer electronics are forcing a rapid increase in the complexity of multiprocessor systems on chip (MPSoCs). Following this trend, MPSoCs are becoming increasingly dynamic and adaptive, for several reasons. One of these is that applications are getting intrinsically dynamic. A streaming application, for instance, can lower its frame rate if the battery charge of a portable device is running low. Another reason is that the workload on emerging MPSoCs cannot be predicted because modern systems are open to new incoming applications at run time. A third reason which calls for adaptivity is the decreasing component reliability associated with technology scaling. Components below the 32-nm node are more inclined to temporal or even permanent faults. In case of a malfunctioning system component, the rest of the system is supposed to take over its tasks.

In our view, the system adaptivity goal shall influence several design decisions, which we list below.

(1) The applications should be specified such that system adaptivity can be easily supported. To this end, we consider Polyhedral Process Networks (PPNs) [1], a special class of Kahn Process Networks (KPNs) [2], as model of computation to specify applications. PPNs are composed by concurrent and autonomous processes that communicate between each other using bounded FIFO channels. Moreover, in PPNs, 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.

Most importantly for our goal, the simple operational semantics of PPNs allows for an easy adoption of system adaptivity mechanisms. For instance, the process state which has to be transferred upon process migration does not have to be specified by hand by the designer and can be smaller compared to other solutions.

(2) As a second design decision, the hardware platform should guarantee the flexibility that adaptivity mechanisms require. Networks-on-Chip (NoCs) [3], which is the platform model considered in our work, are emerging communication infrastructures for MPSoCs that, among many other advantages, allow for system adaptivity. This is because NoCs are generic, since the same 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 PPN model of computation (MoC). Therefore, in this paper, we investigate and propose several communication approaches to overcome this mismatch. All of the proposed approaches consider system adaptivity as a driving objective, and no specific hardware support is required from the platform to realize the inter-tile communication between processes.

(3) Finally, the system must be able to change the process mapping at run-time, using process migration. To this end, we propose and evaluate a process migration mechanism which takes into account specific requirements of the embedded domain such as predictability and efficiency. The efficiency of the proposed process migration mechanism depends on the design decisions discussed above, such as the MoC used to specify the applications. In this respect, the adoption of the PPN MoC ease the realization of process migration in our approach. In our opinion, the problem of a predictable and efficient process migration mechanism in distributed-memory MPSoCs has not received sufficient attention. The aim of our work presented in this paper is to contribute to a more mature solution of this problem.

1.1. Paper Contributions. The contributions of this paper are twofold. On the one hand, we propose and evaluate different communication approaches that implement the PPN semantics on NoC-based MPSoC platforms, enabling mapping-independent and efficient execution of PPN applications, as well as easy process migration. The proposed communication approaches are generic, since they do not rely on specific hardware support from the NoC, and are used to cope with the mismatch between the PPN MoC and the NoC hardware structure.

On the other hand, we develop a predictable process migration mechanism that allows run-time process remapping among the tiles of the NoC, which is a fundamental requirement for system adaptivity. The peculiarity of our solution is that, leveraging the PPN operational semantics and process structure, the migration can actually start at any point during the execution of the main body of a process without the need of moving a large state. Moreover, an upper bound of the process migration overhead can be found, based on the PPN topology and FIFO buffer sizes.

1.2. Related Work. Run-time resource management is a known topic in general purpose distributed systems scheduling [4]. In particular, process migration mechanisms [5, 6], have been developed and evaluated in this context to enable dynamic load distribution, fault resilience, and improved system administration and data access locality. In recent years, run-time management has been gaining popularity and applications also in multiprocessor embedded systems. This domain imposes tight constraints, such as cost, power, and predictability, that run-time management and process migration mechanisms must consider carefully. [7] provides a survey of run-time management examples in state-of-the-art academic and industrial solutions, together with a generic description of run-time manager features and design space.

Our work is focused on a specific component of run-time management strategies, namely, the process migration mechanism. Papers addressing process (or task) migration implementation in MPSoCs can also be found in the literature. The closest to our work is [8], in which the goals of scalability and system adaptivity are achieved through a distributed task migration decision policy over a purely distributed-memory multiprocessor. Similar to our approach, their platform is programmed using a process network MoC. However, in their approach, the actual task migration can take place only at fixed points, which correspond to the communication primitive calls. Our approach, instead, enables migration at any point in the execution of the main body of processes. This leads to a faster response time to migration decisions, which is preferable for instance in case of faults.

Other task migration approaches are explained and quantitatively evaluated in [9, 10]. Dynamic task re-mapping is achieved at user-level or middleware/OS level, respectively. In both these approaches, the user needs to define checkpoints in the code where the migration can take place. This can require some manual effort from the designer which is not needed in our approach. Moreover, a relevant difference from our work is the intertask communication realization, which exploits a shared memory system. We argue that our approach, which uses purely distributed memory, can perform better in emerging MPSoC platforms since it provides better scalability.

The model of computation adopted in our work (Polyhedral Process Networks [1]) not only eases significantly the implementation of system adaptivity mechanism, but it also has several other advantages and applications which can be found in the literature. In particular, our approach exploits the pn compiler [11] to automatically convert static affine nested-loop programs (SANLPs) to parallel PPN specifications and to determine the buffer sizes that guarantee deadlock-free execution. Thus, using the PPN model of computation allows us to program an MPSoC in a systematic and automated way. Although the pn compiler imposes some restrictions on the specification of the input application, we note that a large set of streaming applications can be effectively specified as SANLPs. In addition to the case studies considered in this paper, more application examples regard image/video-processing (JPEG2000, H.264), sound processing (FM radio, MP3), and scientific computation (QR decomposition, stencil, finite-difference time-domain).

Moreover, a recent work [12] has shown that most of the streaming applications can be specified using the Synchronous Data Flow (SDF), model [13]. The PPN model is more expressive than SDF; thus it can as well be used effectively to model most streaming applications.

In general Kahn Process Networks (KPNs), of which PPNs represent a special class, are a widely studied distributed model of computation. They are 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 [14–16], and which aim at supporting and optimizing the mapping of KPN processes on the nodes of a reference platform [17, 18]. In [14, 15], 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 multiprocessor systems [16, 18], or architecture synthesis for FPGA platforms [14] is implemented. A survey of design flows based on the KPN MoC can be found in [19].

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 (possibly heterogeneous) cores well interconnected with a NoC. We provide a PPN API for this platform that the PPN application processes will comply to. Most importantly, the application code remains the same in all possible mappings of the processes. This is achieved by a proposed intermediate layer, called *middleware*, that includes the mapping-related information and implements the PPN communication API.

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 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) [20] 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 [21]. However, these MPI standards are not quite fit for the KPN semantics [22], and building the semantics on top of their primitives brings an overhead compared to platforms with dedicated FIFO support.

The communication and synchronization problem when implementing KPNs over multiprocessor platforms without hardware support for FIFO buffers has been considered in [18, 23]. In [23] the *receiver-initiated* method has been proposed and evaluated for the Cell BE platform. On the same hardware platform, [18] 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 process network semantics, and that we deal with a different kind of platform, with no remote memory access support. Moreover, in both [18, 23], system adaptivity is not taken into account.

In [22] the active *virtual connector* approach has been proposed and evaluated analytically, whereas our results are obtained by the experiments with the real implementation. Moreover, in this paper, we propose yet another approach, namely, *virtual connector with variable rate*.

In [24] 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 since the middleware enables run-time management and the platform is generic; that is, it allows the execution of any application specified as a PPN.

An approach to guarantee blocking write behavior is also used in [8]. 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. The communication approaches described in our paper assume a more proactive behavior of the consumer processes to guarantee the blocking on write compared to the request/acknowledge protocol. We argue that our approach can lead to better performance since it requires less synchronization points.

The remainder of the paper is organized as follows. The solution approach and its main component, the proposed middleware, which performs inter-tile communication and process migration, are introduced in Section 2. The details of the two main middleware parts are described separately, in Section 3 for the inter-tile communication realization and in Section 4 for the process migration mechanism. The applications and case studies used to evaluate the middleware components for inter-tile communication and the process migration mechanism are explained in Section 5, followed by the experimental setup and results. Finally, Section 6 concludes the paper.

2. Proposed Approach

The starting assumption of our system adaptivity approach, as depicted in the right part of Figure 1, is that we target an MPSoC composed of tiles, connected by a NoC, with completely distributed memories and no direct remote memory access. This means that the processing element of a tile can only directly access the content of its own local memory. All the communication and synchronization between processes mapped on different tiles can only happen using messages sent over the NoC.

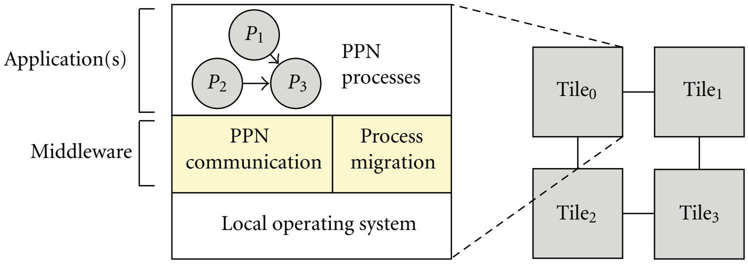


Figure 1: Software infrastructure for each tile of the NoC.

Our approach for realizing system adaptivity consists of deploying the processes of the application(s) modeled as PPNs over the NoC-based MPSoC and allowing their run-time remapping to adapt the system to the changing operating conditions such as variation in quality of service requirements, availability of resources, or power budget constraints. In particular, system adaptivity in our system is supported by using a dedicated middleware, which is highlighted in the software infrastructure diagram in the left part of Figure 1.

At the top of the software stack, applications are described by PPN processes implemented as separate threads. An example of a thread representing a PPN process is given in Figure 3(b), and it will be described in detail in Section 3. However, in this work, the basic structure of PPN processes has been adapted to ease the realization of a predictable process migration mechanism, as will be described in Section 4.

At the bottom of the software stack, the operating system (OS) is responsible for all kinds of process management (process creation, deletion, setting its priority, suspending, or resuming it). These features are essential for the run-time management of the system, and in particular for the execution of process migrations. Moreover, each processor has multitasking capabilities thanks to the OS. In case of *many-to-one* mapping, that is 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/writing from/to a FIFO buffer. When the process blocks, it yields the processor control to the next process in the ready queue in a round-robin fashion.

In between the applications and the operating system, we devised and implemented a middleware which comprises two main components. The first one is the *PPN communication API*, which realizes the communication and synchronization between processes located in separate tiles, according to the PPN semantics. The second one is the *process migration API*, which deals with process creation/deletion, state migration, and the other actions needed for run-time process remapping. The two middleware components will be described thoroughly in Sections 3 and 4, respectively.

3. PPN Communication

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

3.1. Some Definitions. A PPN is a graph defined as a tuple $(\mathcal{P}, \mathcal{C})$, where

(i) $\mathcal{P} = \{\mathrm{P}_1, \ldots, P_N\}$ is a set of processes;

(ii) $\mathcal{C} = \{\mathrm{ch}_1, \ldots, \mathrm{ch}_K\}$ is a set of FIFO channels.

Each process $P \in \mathcal{P}$ has a set of input channels IC_P and output channels OC_P. The processes which write into IC_P are the *predecessors*, and the processes which read from OC_P are the *successors*. The processing element (PE) onto which the process is mapped is denoted as $\mathrm{map}(P)$.

For each channel $\mathrm{ch} \in \mathcal{C}$:

(i) we can derive, using the pn compiler [11], a buffer size B which guarantees deadlock-free execution of the PPN;

(ii) the producer process, which writes data to the channel, and the consumer process, which reads data from it, are denoted, respectively, as $\mathrm{prod(ch)}$ and $\mathrm{cons(ch)}$.

PPN processes communicate and synchronize using these FIFO channels. The PPN semantics forces a process to *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.

All PPN processes have the same code structure, an example of which is given in Figure 3(b). Nested loops iterate, for a given number of times, the body of the process, which is split in three main parts. First, the process reads the input data tokens from (a subset of) the input channels. This is represented by the *read* statements in the figure. Second, the process function (F) produces the output tokens by processing the input tokens. Finally, the output tokens are written to (a subset of) the output channels (*write* statement).

The simplicity of the PPN process structure and semantics eases the development of system adaptivity support, as will be described further in the paper. Only minor changes to the PPN process structure are needed to allow a predictable process migration mechanism, as will be described in Section 4.

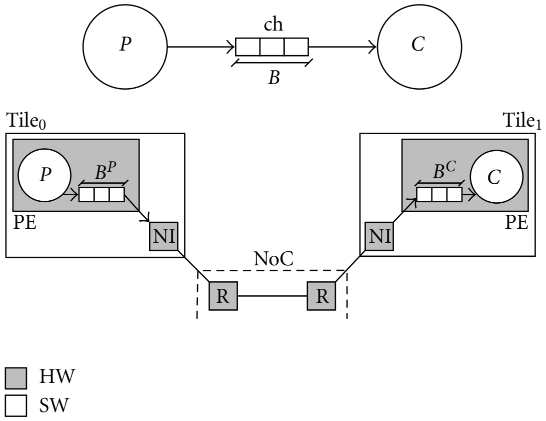


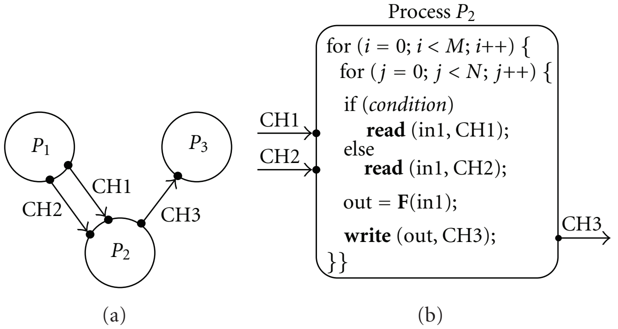
Figure 2: Producer-consumer pair with FIFO buffer split over two tiles.

3.2. Inter-Tile Synchronization Problem. The main problem addressed in this section is the efficient implementation of a communication API allowing the execution of applications modeled as PPNs on Network-on-Chip MPSoC platforms. The first requirement is that this API must respect the PPN semantics. Moreover, we want our middleware to be application-independent and oriented to system adaptivity.

The communication and synchronization problem when mapping PPNs on a NoC is depicted in Figure 2. Consider a producer P and a consumer C connected through an asynchronous communication FIFO buffer B. If both the producer and the consumer can directly access the status register of this FIFO buffer, to check whether it is empty or full, implementing the PPN 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 PPN semantics without a dedicated support from the underlying architecture that allows checking for the status of the remote queues. 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 set such that size(B^P) + size(B^C) $\geq$ size(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 PPN.

Consider again Figure 2. Even if the consumer process C can only access the status of B^C, implementing the *blocking read* is trivial because every time process 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 process B 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 the remainder of this section.



```
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);
}}
```

Figure 3: Example of a PPN (a) and structure of process $P2$ (b).

Furthermore, we want the communication API 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 PPN application processes, an example of which is shown in Figure 3(b). In particular, we want the communication primitives (*read*, *write*) of PPN processes to remain generic, without the notion of process mapping or platform details. These generic primitives are then translated by the communication API implementation in mapping- and platform-dependent function calls.

In all of the communication 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 PPN graph. For instance, when a process is up to send a packet to the consumer via a specific channel, the implementation of the *write* primitive will check in the middleware 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. As described in Section 4, these middleware tables are updated at run-time to allow runtime remapping of application processes over the tiles.

3.3. Virtual Connector Approach (VC). In the virtual connector communication approach, which is depicted in Figure 4, for every channel in the original PPN 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 [22], to the needs of our system implementation. In that work the proposed communication middleware is *active*, meaning that it is implemented using separate threads which deal with the PPN 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.

For each channel in the original PPN graph, we instantiate a software FIFO buffer on the consumer tile. The sizes

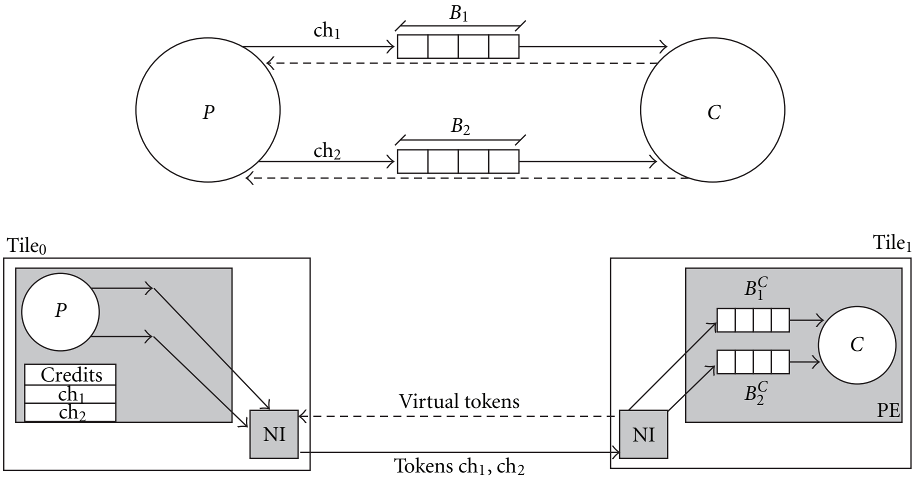
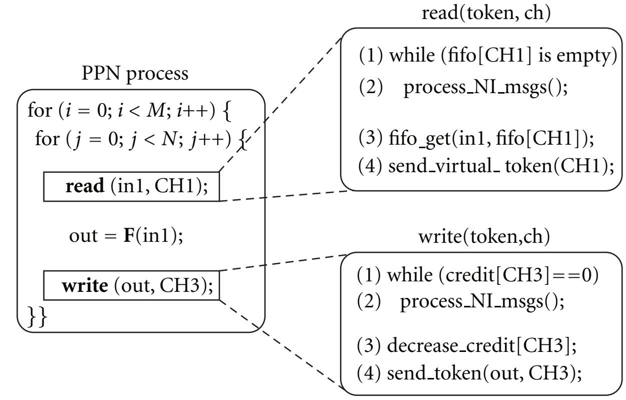


Figure 4: Producer-consumer pair using the virtual connector approach.



```
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) while (fifo[CH1] is empty)
(2)     process_NI_msgs();
(3) fifo_get(in1, fifo[CH1]);
(4) send_virtual_ token(CH1);
```

```
write(token,ch)
(1) while (credit[CH3]==0)
(2)     process_NI_msgs();
(3) decrease_credit[CH3];
(4) send_token(out, CH3);
```

Figure 5: Pseudocode of the VC approach.

of these buffers are 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. This is due to the fact that the credits system guarantees that enough locations are free on the remote buffers before sending a token. Therefore, referring back to Figure 2, 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. 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. On 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. In this way the producer is constantly updated about the status of the remote FIFO buffers.

The pseudocode of the VC approach is shown in Figure 5. 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 increases the credit of channel j.

In Figure 5, 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 Figure 12.

Similarly, in the *write* primitive in Figure 5, 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 Figure 12).

3.4. 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 for all $i \in \{1, \ldots, N_{\text{ch}}\}$ $1 \le n_i \le \text{size}(B_i)$, where N_{ch} represents the number of channels in the PPN graph. The *credit* variable for

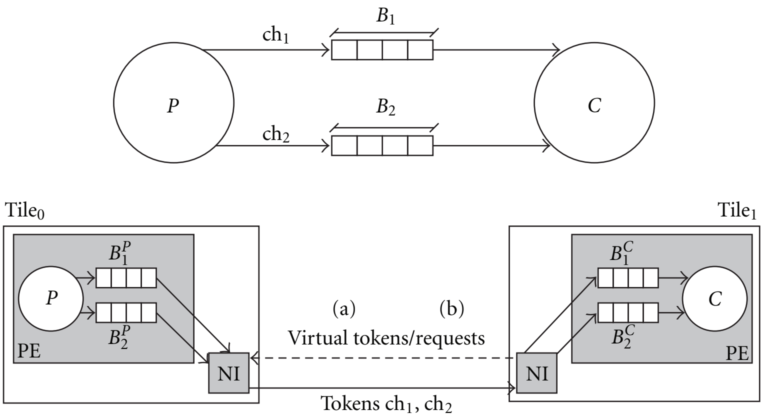


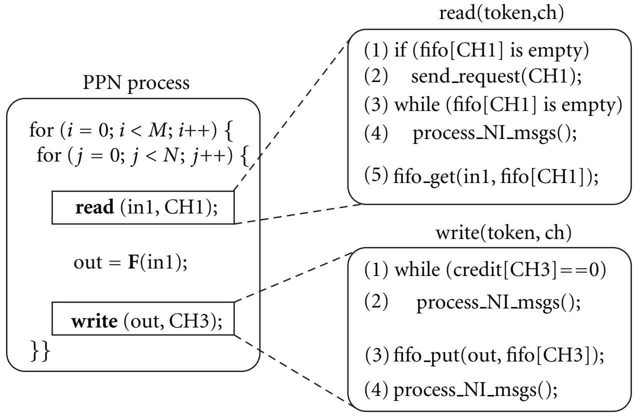
FIGURE 6: Producer-consumer implementation: when using the VRVC, the producer receives back virtual tokens (a); when using R, it receives requests (b).

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 PPN 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 buffer 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 PPN graph. This means that for all $i \in \{1,\ldots,N_{\mathrm{ch}}\}$ $\mathrm{size}(B_i^P) = \mathrm{size}(B_i^C) = \mathrm{size}(B_i)$, as depicted in Figure 6, case (a). The pseudocode for the VRVC approach is omitted for the sake of brevity.

3.5. Request-Driven Approach (R). This method is very similar to the approach used in [23] for realizing the FIFO communication on the Cell BE platform. 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 PPN graph (B_i), such that for all $i \in \{1,\ldots,N_{\mathrm{ch}}\}$ $\mathrm{size}(B_i^P) = \mathrm{size}(B_i^C) = \mathrm{size}(B_i)$. 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 Figure 6, 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.



```
PPN 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 (credit[CH3]==0)
(2)     process_NI_msgs();
(3) fifo_put(out, fifo[CH3]);
(4) process_NI_msgs();
```

FIGURE 7: Pseudocode of the R approach.

Figure 7 shows the pseudocode of this communication approach. Similarly to the VC approach, 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 in Figure 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 packet of tokens for the calling channel arrives. The *blocking on write* is implemented in lines 1-2 of the write primitive in Figure 7. 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.

4. Process Migration

This section provides a description of the proposed PPN process migration mechanism over the NoC-based MPSoC

Table 1: Middleware table example.

ch	prod(ch), cons(ch)	map(prod(ch)), map(cons(ch))
1	P_1, P_2	Tile_0, Tile_1
2	P_2, P_3	Tile_1, Tile_2

system. It is a fundamental part of the middleware depicted in Figure 1 because it realizes the run-time remapping of processes, which in turn allows system adaptivity strategies.

The migration mechanism depends on the considered communication approach. As a starting assumption to devise the migration mechanism, we consider the *request-driven* (R) communication approach described in Section 3.5. This choice is made because the R approach leads to a considerably easier implementation of the migration mechanism since it requires less synchronization points. At the same time, it gives performance comparable to the other approaches for computation-dominant applications, as will be shown in Section 5.

We recall that to take into account the run-time remapping of processes over the NoC, each PE stores in its local memory a *middleware table* which is used to refine the generic communication primitives to mapping-dependent function calls. An example of a middleware table generated for the initial mapping in Figure 8 is given in Table 1. For each channel of the PPN, the producer and consumer process IDs are stored, together with their current mapping in the system. Auxiliary information, for instance, pending requests during migration execution, is also saved for each channel.

Mainly two kinds of process migration mechanism can be considered, namely, *process replication* and *process recreation*. In process replication, the program code of a process that can be migrated is copied in each tile, thereby creating replicas of the process. When a process needs to be migrated from one tile to another, the process is suspended on the first tile and restarted on the second. The state of the process must be copied from the first tile to the second because the process cannot be just restarted from scratch.

The second kind of process migration mechanism is based on the so-called *process recreation*. In this case, if a migration is needed, the process is killed on the original tile it runs and created on another tile by moving both the process code and state. The OS/middleware in this case must support dynamic loading of processes to processors. This way, only one instance of the process code exists at a given time in the system.

On the one hand, the process replication mechanism is less efficient in terms of memory usage, compared to the process recreation. On the other hand, it offers significant advantages such as easier implementation and faster migration procedure. We chose the process replication mechanism because we consider the fast execution of process migration more important. Moreover, the memory constraint in our system is not critical.

A simple diagram showing the migration of a PPN process is depicted in Figure 8. Even though this is a simple example, it can be easily generalized for more complex PPN topologies. The diagram highlights the tiles involved in the process migration procedure, which are referred to as:

(i) the *source tile*, namely, the tile which runs the process before the migration takes place,

(ii) the *destination tile*, which is the tile that will execute the process after the migration,

(iii) the *predecessor tile*(*s*), which run(s) the predecessor process(es), and

(iv) the *successor tile*(*s*), which executes the successor process(es).

The structure of PPN processes, modified to allow migration at any point during the execution of the process main bodies, and the proposed process migration mechanism are described in the following two subsections.

4.1. Migratable PPN Process Structure. Our goal is to allow the migration to be performed at any time during the execution of the process main body, in order to improve the migration response time. To this end, we extend the NI interface of a tile with the ability to generate an interrupt for the processing element when a message with a reserved tag is received. This extension is made because the detection of migration decisions by polling at specific migration points in the code may cause undesired latency in the migration procedure.

With the requirement that migration may happen at any point within the execution of the processes main body, we devise the structure of a migratable PPN process as shown in Figure 9. It is based on the structure shown in Figure 3(b), which we will refer to as *basic process structure*.

We comment and motivate the migratable PPN process structure shown in Figure 9 in the following. When the thread starts, in line 1, it checks if the *migration* flag is set. If the checking is positive, it means that a migration has been performed, so the process state is reloaded.

Since the PPN model definition requires a *stateless* process function, for example, F_2 in Figure 9, that is, a function whose execution does not depend on the previous iterations, the state of a PPN process is represented only by:

(i) the content of its input and output FIFOs;

(ii) its iterator set, namely, the values of the nested loop iterator variables, see (i, j) in Figure 9, lines 2-3.

When a function requires to have a state, it is represented in the PPN model by a stateless function with FIFO self-edges, which represent the function state.

Both state components listed above are transferred from the source tile to the destination tile upon migration. If the migration flag is false, it means that the process starts from scratch, with empty input and output FIFOs and $i_0 = j_0 = 0$.

Lines 2 and 3 differ from the basic process structure in Figure 3(b) because the iterators inside the *for* loops do not start from zero in case of migration. Instead, they start from the values i_0 and j_0, which represent the iteration at which the process was interrupted by the migration while running

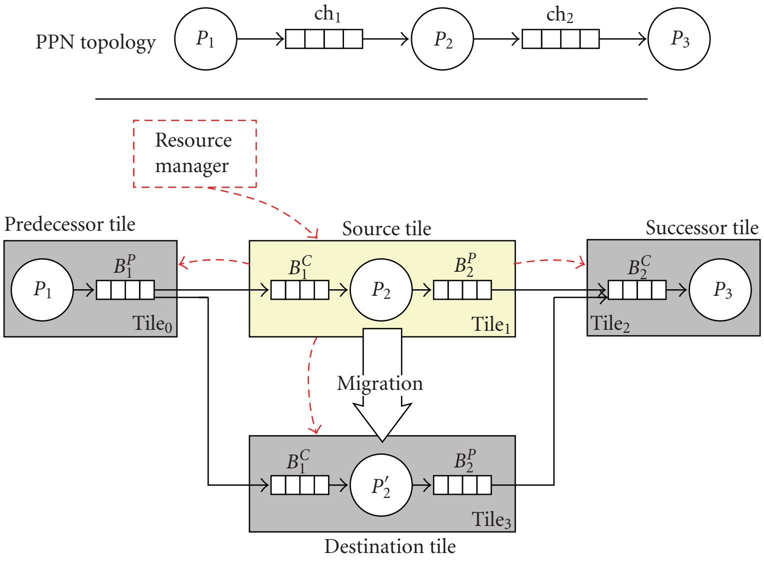
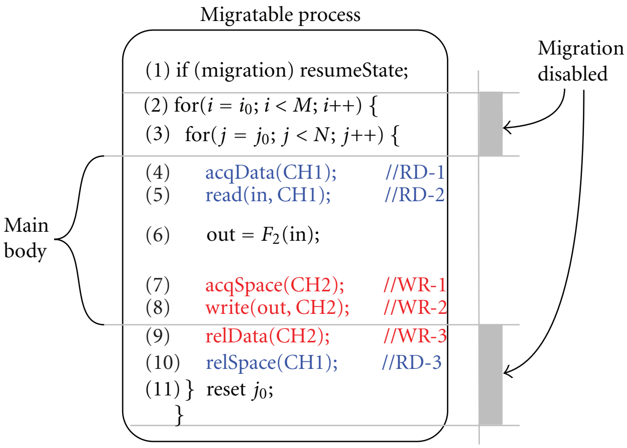


Figure 8: Migration diagram.



```
(1) if (migration) resumeState;
(2) for(i = i0; i < M; i++) {
(3)   for(j = j0; j < N; j++) {
(4)     acqData(CH1);        //RD-1
(5)     read(in, CH1);       //RD-2
(6)     out = F2(in);
(7)     acqSpace(CH2);       //WR-1
(8)     write(out, CH2);     //WR-2
(9)     relData(CH2);        //WR-3
(10)    relSpace(CH1);       //RD-3
(11) }  reset j0;
     }
```

Figure 9: Migratable PPN process.

on the source tile. After the first complete execution of the inner *for* loop, starting from j_0, the value of j_0 is set to zero in line 11 such that the next execution of the inner loop starts correctly with $j = 0$.

The communication primitives are different from the ones used in the basic process structure. The *read* primitive, for instance, is split into three separate operations (see lines 4, 5, and 10). First, the input channel (CH1) is tested to verify the presence of an available data token, using the *acquireData* function (*acqData*(CH1) in line 4). Then, the token is actually copied from the software FIFO to the input variable which will be processed by the process function F_2. The copy operation is performed in line 5. However, differently from the normal *read* primitive, the memory locations occupied by the read token are not released immediately. The actual release, which consumes the data from the FIFO by increasing the read pointer, takes place only in line 10 (*relSpace*(CH1)). This way, if a migration is triggered before the release instruction, the process can be correctly resumed on the destination tile since it will read again the same input token, because the read pointer is not changed. Similarly, the *write* primitive is split in three operations, see lines 7, 8, and 9, of which only *relData* affects the write pointer. Finalizing the *read* and *write* operations at the end of an iteration allows the process migration to happen anywhere within lines 4–8 correctly. Note that, in case of multiple input or output channels, the release operations should be grouped together and placed right after the main body of the process, in order to guarantee a consistent process state.

Process migration cannot happen within the lines 9–11 and 2-3 because that would cause an inconsistency in the migrated process state. This is because lines 9 and 10 can be considered as an update of the output and input FIFOs state, while lines 11, 2, and 3 represent the iterator set update. If, for instance, a migration happens after the FIFO state update but before the iterator set update, the migrated process will restart the execution with the FIFO status corresponding to the next iteration, but with the iterator set of the current (interrupted) iteration. This condition will certainly cause a deadlock. Although the process migration cannot happen within lines 2-3 and 9–11, we note that these sections represent a minimal part of the process execution, because performing the update of read and write pointers and iterator sets is a matter of a few simple instructions. Therefore, disabling the migration within these sections does not increase the response time significantly.

The principle behind the proposed migratable process structure is that the state of a process must be *consistent* and *up-to-date* when a migration is performed. This allows the migrated process to correctly resume its execution on the destination tile. Leveraging the PPN process structure, our approach does not require the designer to specify the context that has to be transferred upon migration as in [9]. This burden is neither moved to the OS/middleware level as in [10]. Determining the state to be migrated is not needed because the PPN process state simply consists of the two components described above. Moreover, our approach does not need designer-generated checkpoints/migration points. The resource manager in Figure 8 can interrupt the process execution at any time during the execution of the process main body. The migrated process will then resume its execution from the beginning of the interrupted iteration. On the one hand, this implies that if the migration is triggered in the middle of the function execution, the time since the start of the iteration is lost. On the other hand, this approach leads to a more efficient implementation and predictable migration response time, which we consider more important for our goals.

4.2. Process Migration Mechanism. The migration mechanism requires actions from all the tiles depicted in Figure 8. The migration decision is taken by the resource manager, which sends a specific control message to the source tile. How the resource manager takes the migration decision is out of the scope of this paper because we focus on the process migration mechanism itself. The source tile then broadcasts this control message to the destination, predecessor, and successor tiles to complete the migration procedure.

The control messages which notify the process migration to the involved tiles contain the ID of the migrated process (*ctrl_msg.migProc_ID*) and the new mapping of that process (*ctrl_msg.dest_PE*). On all of the involved tiles, and on the resource manager, the middleware tables are then updated by executing the following operations, for each channel in the list:

(i) if (*prod(ch)==ctrl_msg.migProc_ID*)
update *map(prod(ch))* to *ctrl_msg.dest_PE*,

(ii) if (*cons(ch)==ctrl_msg.migProc_ID*)
update *map(cons(ch))* to *ctrl_msg.dest_PE.*

For each of the tiles involved in the migration procedure, the detailed list of required actions is explained below.

4.2.1. Actions on the Source Tile. On the source tile, the process has to be stopped, and its state saved and forwarded to the destination tile. Moreover, the middleware table is updated as described above. The source tile takes also care of propagating the migration decision to the other tiles involved in the migration procedure. This propagation is depicted by the dashed arrows in Figure 8.

4.2.2. Actions on the Destination Tile. The destination tile receives a specific message for process activation. The migration procedure is handled by creating the required software FIFOs and by activating the replica of the migrated process using the corresponding OS call. Before the process replica is started, the *migration* flag is set to 1 so that the state of the migrated process is resumed (see line 1 in Figure 9). This implies that the input and output FIFOs of the migrated process are copied, and the iterator set (in the figure, i_0 and j_0) is set such that the execution starts from where it was suspended on the source tile. The middleware table is also updated in the way described above.

4.2.3. Actions on Predecessor Tile(s). On these tiles, the only required step is the update of the middleware tables according to the new mapping of the migrated process. This way, new tokens meant for the migrated PPN process, will be sent to the destination tile.

A corner case of the communication between the migrated process and its predecessors may happen when the process has sent a *request* for new tokens just before the migration command arrives. If that request has been served, it means that new tokens are either traversing the NoC or they are already stored in the source tile. The predecessor tile in this case has to send another interrupt-generating message to the source tile, in order to force the forwarding of these data tokens to the destination tile.

4.2.4. Actions on Successor Tile(s). Similarly, the successor tiles have to update the middleware tables so that the new requests for data tokens will be sent to the destination tile. A particular case in the protocol between successor processes and the migrated process is represented by requests which are sent to the source tile just before the interrupt decision takes place. In this case, if the requests are not served before the migration, they have to be forwarded to the destination tile.

5. Experiments and Results

In order to evaluate the proposed middleware, we perform two experiments to assess both its main components. In the first experiment, described in Section 5.2, we compare the efficiency of the different approaches for the PPN communication API in two case studies. In the second experiment, described in Section 5.3, we assess the process migration benefits and overhead by applying our migration mechanism in one of the case studies. Before presenting these two experiments, we describe the case studies and the experimental setup that we used to obtain the results.

5.1. Case Studies and MPSoC Platform Setup. We evaluate the three communication approaches presented in Section 3 on two applications modeled as PPNs with extremely different communication/computation characteristics. The reason is that we want to compare the overhead of the different approaches between two extremes. The Sobel filter application described in Section 5.1.1 represents the worst case (the first extreme), when the computation/communication ratio is low and the PPN topology is complicated. The M-JPEG encoder application described in Section 5.1.2, on the other extreme, is computation dominant and with relatively simple PPN 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.

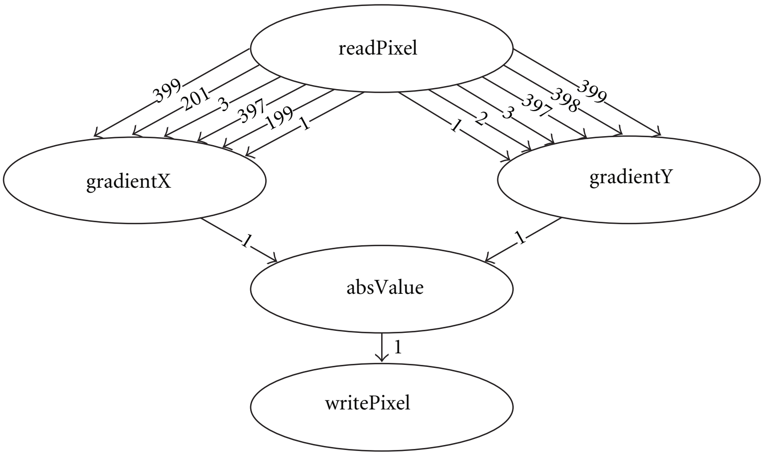


Figure 10: PPN specification of the Sobel filter.

Table 2: Execution times of sobel functions.

Process	Execution time (c.c.)
ReadPixel	5
Gradient X	31
Gradient Y	31
absValue	118
WritePixel	5

Table 3: 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)

5.1.1. Sobel Filter. The Sobel application is an edge-detection algorithm for digital images. Its PPN graph is shown in Figure 10, where the numbers over edges indicate the minimal buffer sizes needed for processing a 200 × 122 pixel input image. The PPN 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 2. 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.

5.1.2. M-JPEG Encoder. The PPN specification of this application is shown in Figure 11. 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 cycles required for the execution of each function of the M-JPEG application are summarized in Table 3. This application shows a much simpler communication and synchronization pattern compared to Sobel, and it also has a much higher computation/communication ratio.

5.1.3. MPSoC Platform Setup. The system on which we evaluated our communication approaches is based on a 2 × 2 mesh of tiles, connected via a Network-on-Chip. Each tile is composed by a MicroBlaze processor, with its local program and data memories, and a network interface. The platform does not support remote memory access.

The network interface contains only two hardware FIFOs, 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 channels of the PPN graph. In the opposite direction, when a packet has to be sent over the NoC, the processor forwards data from its local 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*.

The actual structure of the different kind of messages that are sent over the NoC is represented in Figure 12 for the VC and R communication 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-level packet (denoted as MW-level packets in the figure). The structure of middleware-level packets depends on the communication approach. In the R approach, a request for channel number i is implemented as a single flit, with value $-i$. By contrast, a packet used for transferring tokens 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

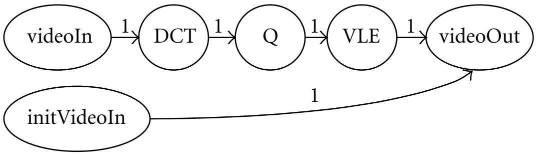


Figure 11: PPN specification of the M-JPEG encoder.

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 middleware-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; that is, *n_tokens* is always equal to one.

5.2. Inter-Tile Communication Efficiency. The MPSoC platform described in Section 5.1 has been implemented on a Virtex5 FPGA prototyping board. We run the two application case studies using all the communication approaches proposed in Section 3 to obtain the results described below. The experiments for process migration are also described later in this section. Note that the proposed migration mechanism is generic, meaning that it is not dependent on a particular NoC implementation.

In order to compare the efficiency of the inter-tile communication of the different communication approaches, we execute the two case study applications with fixed mappings shown in Figure 13. 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 communication approach.

We found out experimentally that the parameter n_i of the VRVC approach gives the best performance when it is set to its maximum value, that is, when for all $i \in \{1,\ldots,N_{\text{ch}}\}$ $n_i = \text{size}(B_i^C)$. The performance results, summarized in Figure 14, show a large difference of the execution time for the Sobel application when using different communication approaches. However, in the M-JPEG case, all of the communication approaches yield to similar performance 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 → software FIFO → 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 communication 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 [14], 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 application modeled as PPN. The performance slowdown, when comparing the NoC system with the point-to-point systems, is shown in Figure 15. It is noticeable that the Sobel application is highly penalized in the execution on our NoC system, whereas 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 PPN channels have to share the same physical channel (the NoC link). The second reason is a consequence of the first one. In the NoC case, the presence of only one physical link, being shared between different PPN channels, poses the need for a flow-control policy. To optimize for low hardware overhead, we chose to implement the control flow at the middleware level, based on software FIFOs on the producer and on the consumer side. This requires additional memory copy operations to dispatch/multiplex the communication tokens to/from the correct software FIFO. Such copies are unnecessary in the case of adoption of multiple point-to-point connections with hardware FIFOs.

Another important metric when executing applications on a NoC-based MPSoC 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, the 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 communication approach that is used. The results, depicted in Figure 16, show two extremes: the VC and R approaches. This large difference can be explained by two factors. The first factor is the overhead of packet headers. On one hand, in the VC approach, since there is no packetization of tokens, each token travels in the NoC with its own header. On the other hand, in the R approach, 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

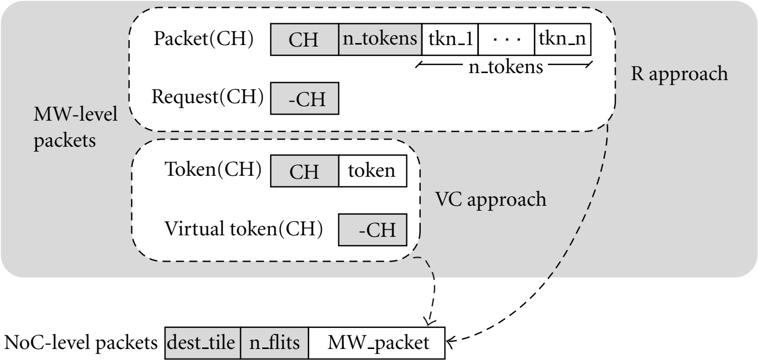
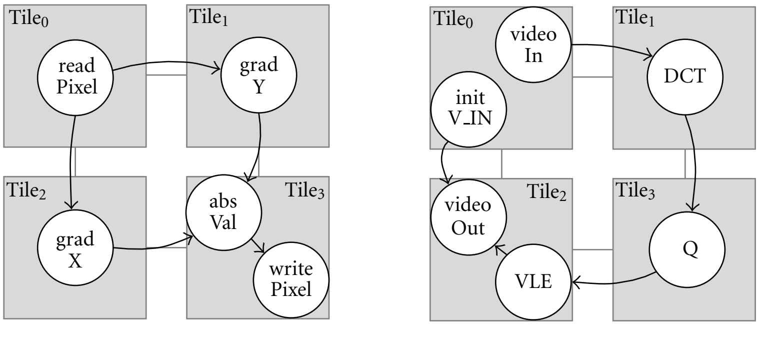


Figure 12: Structure of middleware- and network- level packets.



(a)

(b)

Figure 13: Fixed mappings for Sobel (a) and M-JPEG (b) to test the different communication approaches.

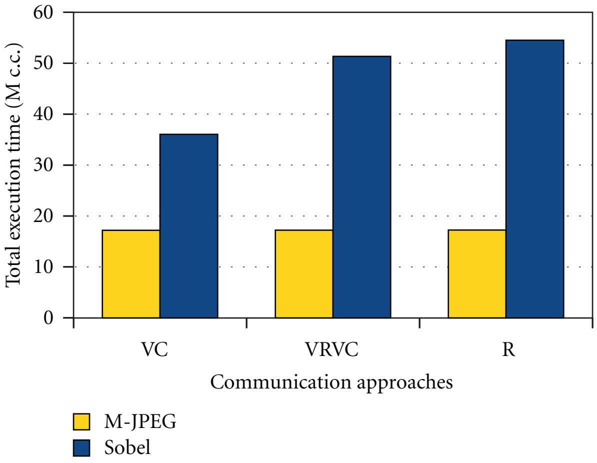


Figure 14: Total execution time for different communication approaches.

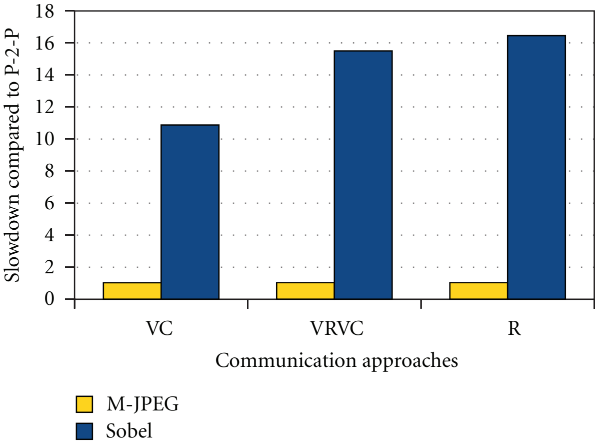


Figure 15: Slowdown for different communication approaches.

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, whereas in the *R* approach the requests are made less frequently, just when the consumer is blocked on reading.

5.3. Process Migration Benefits and Overhead. System adaptivity requires the ability to change the process mapping at run-time in a predictable and efficient way. To illustrate the benefits of our migration approach presented in Section 4, we compare our proposed migration mechanism, driven by interrupt-generating control messages, with a migration approach based on migration points.

In the latter case, process migration can take place only at fixed points in the code. The setup of this experiment is

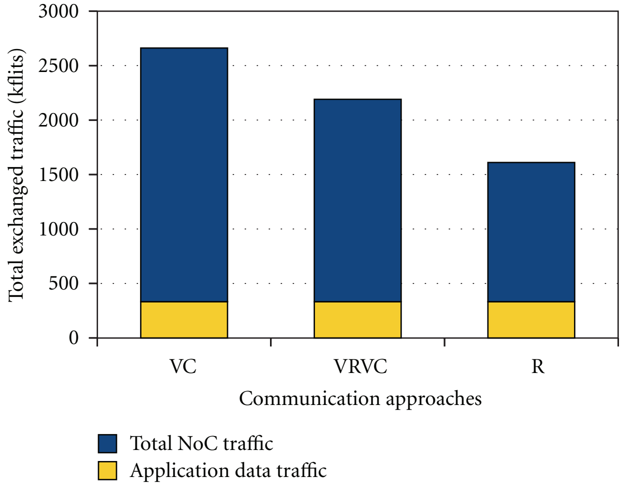


Figure 16: Traffic injected into the NoC by executing Sobel with different communication approaches.

shown in the left part of Figure 18. We use as a case study the M-JPEG application described in Section 5.1.2. Tile_1 initially runs all of M-JPEG processes, which are listed in Figure 19. P_1 is derived by merging *initVideoIn* and *videoIn* processes, P_2 and P_3 represent, respectively, the *DCT* and *Q* processes, and P_4 is obtained by merging the *VLE* and *videoOut* processes. We use the M-JPEG application as a case study because, compared to the Sobel application, in M-JPEG processes are coarse grained with high computation/communication ratio, and therefore M-JPEG represents better the kind of applications which are likely to be mapped on a NoC-based MPSoC. The scheduling of the M-JPEG processes on Tile_1 before the migration is represented in Figure 17. Scheduling charts have been obtained using the GRASP [25] trace visualization tool to plot the information gathered at run time. The trace shows the periodic scheduling which is executed when all the processes are mapped on one tile and the scheduling policy is data driven. The buffer size of all the FIFO channels is set to two in this experiment. In this scenario, the process scheduling iterates in the following way. First, P_1 executes two times, until it blocks on writing because its output buffer is full. Then P_2 is scheduled. It completes two iterations, consuming the tokens created by P_1 and producing two tokens for P_3. It then blocks while reading its input FIFO which is empty by then. Similarly, P_3 and P_4 execute twice before blocking on read. This scheduling repeats until the end of the application execution if no migration is performed.

In Figure 17, the arrows over the bars of process P_1 represent the start of an iteration of that process (for the sake of clarity, see line 4 in Figure 9). Assume that these points correspond to migration points, namely, where the process checks if migration messages have been sent by the resource manager. Given that the migration request can reach Tile_1 at any time, the latency of the actual process migration can vary. In the best case, the migration request reaches the tile right before a migration point. In the worst case, the migration request arrives just after a migration point, for instance, the one which is reached around clock cycle 275,000. The actual migration would not take place until the next migration point, which happens to be after 2 executions of P_3, P_4, and P_1, and one execution of P_2. In this simple case, an upper bound of the process migration response time can be found, based on the process scheduling, which in turn depends on the workload of processes, the buffer sizes, and the scheduling policy. In more complex cases, where the scheduling on one tile is affected by the scheduling on other tiles because of data dependencies, even finding an upper bound for the response time practically would not be possible.

By contrast, the interrupt-driven migration mechanism that we propose in Section 4 has a predictable behavior. As shown in Figure 18, the system has a faster response time to migration requests. At time τ_1, which is the worst case for the fixed-point migration strategy discussed above, the resource manager sends a control message which triggers the migration of P_2 to Tile_2. The process can be restarted on the destination tile within a predictable amount of time represented by the difference $(\tau_1 - \tau_2)$. This is the time it takes the source tile and the destination tile to execute the steps described in Section 4, such as the movement of the process state and the activation of process P_2 on the destination tile. This migration overhead in time $(\tau_1 - \tau_2)$, as shown in Figure 18, is way smaller than a single execution of the DCT function in process P_2. The migration procedure in this example actually takes less than 12% of a single execution of the DCT process.

Note that an upper bound of the migration procedure overhead can be derived for guaranteed throughput (GT) NoCs. In fact, the migration duration T_{mig} of a process $P \in \mathcal{P}$ can be split in two main components:

$$T_{\text{mig}}(P) = T_{\text{stateMig}}(\text{stateSize}(P)\ ) + T_{\text{procAct}}, \tag{1}$$

T_{procAct} is a constant value which represents the time required to activate the migrated process using OS system calls, to update the middleware table, and complete all the actions described in Section 4 on the destination tile. T_{stateMig} is the time it takes to transfer the state from the source to the destination tile. Its worst case, for GT NoCs, depends only on the state size. The largest state size of a process P is obtained when both the input and output FIFO buffers of P are full. This worst-case value can then be derived from the PPN topology and buffer sizes

$$\max(\text{stateSize}(P)) = \sum_{\text{ch}\in\text{IOC}_P} \text{size}(B(\text{ch})), \tag{2}$$

where $\text{IOC}_P = \text{IC}_P \cup \text{OC}_P$ as defined in Section 3.1, $\text{size}(B(\text{ch}))$ is the size of the buffer which represents the channel ch on the source tile. The value $\text{size}(B(\text{ch}))$ is obtained by multiplying the number of tokens of $B(\text{ch})$ by the token size of a channel ch. An upper bound of the migration time T_{mig} of a process P can be calculated using $\max(\text{stateSize}(P))$ in (1).

The worst case, for our interrupt-driven migration mechanism, is represented by the arrival of a migration request just before the end of a function execution in a

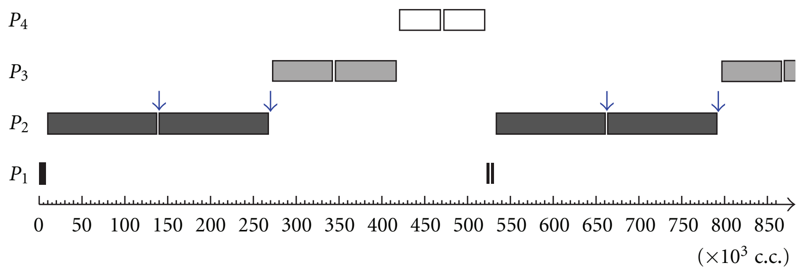


Figure 17: M-JPEG process scheduling when running on a single tile.

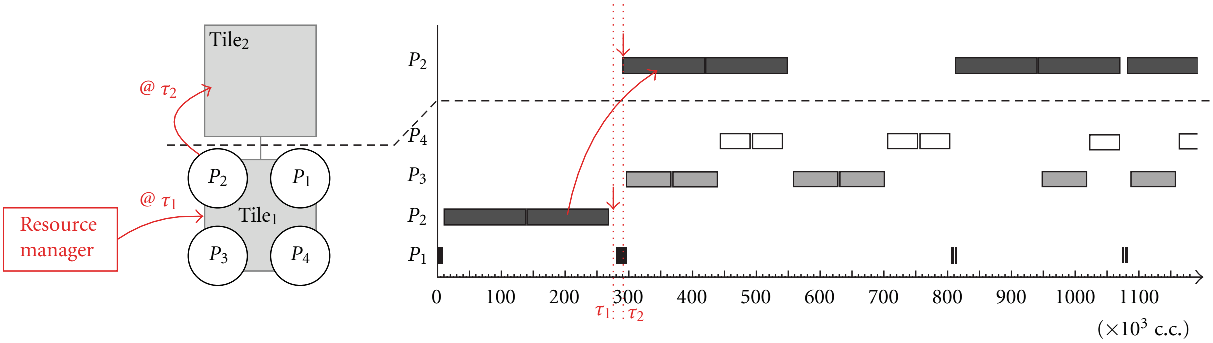


Figure 18: M-JPEG process scheduling while migrating P_2 using the proposed migration mechanism.

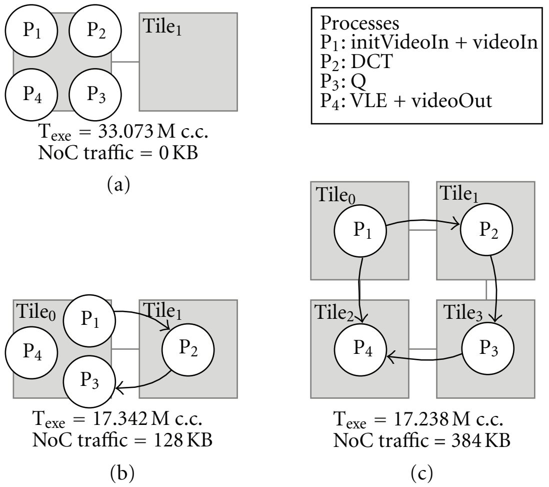


Figure 19: Execution time and generated traffic as a function of the process mapping. Only inter-tile communication links are depicted.

process that has to be migrated. In this case, the migration still takes place in a predictable amount of time but the process execution has to roll back to the beginning of the interrupted iteration. All the time spent in the function execution is wasted in this scenario.

The proposed process migration mechanism allows our system to change its configuration at run time. The resource manager triggers process migrations such that the system dynamically moves from the configuration (a) to (b), then to (c) in Figure 19. By doing this, the resource manager is capable of changing, at run time, the total execution time (T_{exe} in Figure 19), and total exchanged traffic over the NoC (*NoC traffic* in Figure 19). Both T_{exe} and *NoC traffic* correspond to the processing of a single input frame using the M-JPEG application. The resource manager, for instance, can decide to change system configuration because of a quality of service requirement demanded at run-time by the user.

In detail, in Figure 19(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. In this configuration, the execution time per one frame is $T_{\text{exe}} = 33.073$ millions of clock cycles. However, in Figure 19(b), processes P_1, P_3, and P_4 are executed on one tile, and process P_2 runs on a separate tile. Since process P_2 (the DCT) is the most computationally intensive in M-JPEG, accounting for 51% of the total workload, the obtained speedup compared to (a) is close to 2. In fact the execution time per frame drops to $T_{\text{exe}} = 17.342$ millions of clock cycles. The mapping in Figure 19(c) does not show a relevant further performance improvement because P_2 represents the bottleneck of the M-JPEG application, such that even just migrating it to a separate tile as in (b) gives almost optimal performance. This experiment shows the efficiency of the proposed process migration procedure. The system is allowed to substantially change its execution time per frame, at run time, with an almost negligible overhead. As explained above, the process migration overhead is way smaller than a *single* execution of the DCT process. The negligible performance speedup obtained when changing the system configuration from (b) to (c) does not depend on the migration overhead. It is actually caused only by the intrinsic structure of the M-JPEG application.

6. Conclusions

This paper proposes a middleware support for the execution of polyhedral process networks on Network-on-Chip MPSoCs allowing system adaptivity. Two main middleware components have been devised and implemented.

The first one is the *PPN communication API*, which realizes the PPN semantics on NoC implementations. Three communication approaches have been evaluated experimentally on two applications with very different computation and communication characteristics. The results show 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, if compared to customized point-to-point systems. On the contrary, when the computation/communication ratio of an application is higher, the overhead introduced by the execution on NoC with all the proposed communication approaches is much lower.

The second middleware component concerns the process migration procedure, which is essential for system adaptivity. A reactive and predictable process migration mechanism has been devised and developed. The proposed mechanism does not need user-specified checkpointing since it exploits the simple structure of PPN processes, whose state is only represented by iterator sets and the content of input/output FIFO buffers. Moreover, it allows the execution of a migration at any time during the execution of the main body of processes, since it does not rely on fixed migration points. The proposed migration mechanism is predictable because an upper bound of its overhead can be derived, for GT NoCs, from the process network topology and buffer sizes. Moreover, we show using the M-JPEG application case study that the migration mechanism allows the system to change its performance metrics at run time with almost negligible overhead.

Acknowledgments

This work was funded by the European Commission under the Project MADNESS (no. FP7-ICT-2009-4-248424) and by the ARTEMIS JTU project ASAM (ART CALL 2009 100265). Moreover, researchers involved in this paper received a collaboration grant by the HiPEAC Network of Excellence and two grants by Regione Sardinia: Master and Back and Young Researchers Grant, PO Sardegna FSE 2007/2013, L.R.7/2007, CRP1 166, "Promotion of the scientific research and technological innovation in Sardinia". The paper reflects only the authors' view; the funding institutions are not liable for any use that may be made of the information contained herein. The preliminary results and an initial version of some of the ideas presented in this paper have been previously described in the paper "Middleware Approaches for Adaptivity of Kahn Process Networks on Networks-on-Chip," to appear in the Proceedings of the 2011 Conference on Design and Architectures for Signal and Image Processing (DASIP 2011), November 2–4, Tampere, Finland.

References

[1] S. Verdoolaege, "Polyhedral Process Networks," in *Handbook on Signal Processing Systems*, Springer, 2010.

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

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

[4] T. L. Casavant and J. G. Kuhl, "A taxonomy of scheduling in generalpurpose distributed computing systems," *IEEE Transactions on Software Engineering*, vol. 14, no. 2, pp. 141–154, 1988.

[5] J. M. Smith, "A survey of process migration mechanisms," *SIGOPS Operating Systems Review*, vol. 22, pp. 28–40.

[6] D. S. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process migration," *ACM Computing Surveys*, vol. 32, no. 3, pp. 241–299, 2000.

[7] V. Nollet, D. Verkest, and H. Corporaal, "A safari through the MPSoC run-time management jungle," *Journal of Signal Processing Systems*, vol. 60, no. 2, pp. 251–268, 2010.

[8] G. M. Almeida, G. Sassatelli, P. Benoit et al., "An adaptive message passing MPSoC framework," *International Journal of Reconfigurable Computing*, vol. 2009, Article ID 242981, 20 pages, 2009.

[9] S. Bertozzi, A. Acquaviva, D. Bertozzi, and A. Poggiali, "Supporting task migration in multi-processor systems-on-chip: a feasibility study," in *Proceedings of the Design, Automation and Test in Europe (DATE '06)*, pp. 15–20, March 2006.

[10] A. Acquaviva, A. Alimonda, S. Carta, and M. Pittau, "Assessing task migration impact on embedded soft real-time streaming multimedia applications," *Eurasip Journal on Embedded Systems*, vol. 2008, Article ID 518904, 15 pages, 2008.

[11] S. Verdoolaege, H. Nikolov, and T. Stefanov, "pn: a tool for improved derivation of process networks," *Eurasip Journal on Embedded Systems*, vol. 2007, Article ID 75947, 13 pages, 2007.

[12] W. Thies and S. Amarasinghe, "An empirical characterization of stream programs and its implications for language and compiler design," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pp. 365–376, 2010.

[13] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

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

[15] A. Nieuwland, J. Kang, O. P. Gangwal et al., "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, no. 3, pp. 233–270, 2002.

[16] S. Kwon, Y. Kim, W. C. Jeun, S. Ha, and Y. Paek, "A retargetable parallel-programming framework for MPSoC,"

ACM Transactions on Design Automation of Electronic Systems, vol. 13, no. 3, pp. 1–18, 2008.

[17] 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., pp. 1007–1040, Springer, 2010.

[18] W. Haid, L. Schor, K. Huang, I. Bacivarov, and L. Thiele, "Efficient execution of Kahn Process Networks on multiprocessor systems using protothreads and windowed FIFOs," in *Proceedings of the IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia '09)*, pp. 35–44, IEEE, Grenoble, France, 2009.

[19] W. Haid, K. Huang, I. Bacivarov, and L. Thiele, "Multiprocessor SoC software design flows: a focus on Kahn process networks," *IEEE Signal Processing Magazine*, vol. 26, no. 6, pp. 64–71, 2009.

[20] "Multicore associations communication API," http://www.multicore-association.org/.

[21] "A high performance message passing library," http://www.open-mpi.org/.

[22] 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, Article ID 295385, 14 pages, 2011.

[23] D. Nadezhkin, S. Meijer, T. Stefanov, and E. Deprettere, "Realizing FIFO communication when mapping Kahn process networks onto the cell," in *Proceedings of the Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS '09)*, pp. 308–317, Springer, Berlin, Germany, 2009.

[24] A. B. Nejad, K. Goossens, J. Walters, and B. Kienhuis, "Mapping KPN models of streaming applications on a network-on-chip platform," in *Proceedings of the Workshop on Signal Processing, Integrated Systems and Circuits (ProRISC '09)*, November 2009.

[25] M. Holenderski, M. M. van den Heuvel, R. J. Bril, and J. J. Lukkien, "Grasp: tracing, visualizing and measuring the behavior of real-time systems," in *Proceedings of the International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS '10)*, July 2010.

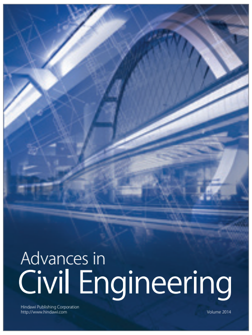
Journal of
Engineering

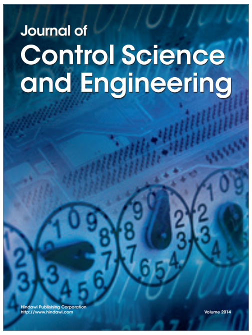
The Scientific
World Journal

International Journal of
Rotating
Machinery

Journal of
Sensors

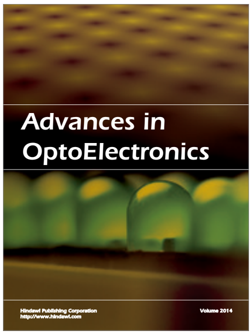
International Journal of
Distributed
Sensor Networks


Advances in
Civil Engineering


Journal of
Control Science
and Engineering

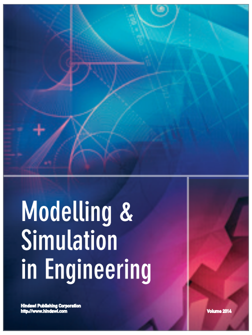
Journal of
Robotics

Journal of
Electrical and Computer
Engineering

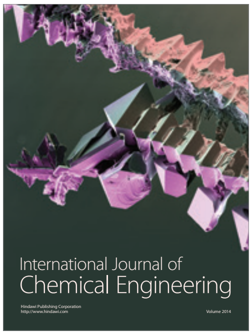

Advances in
OptoElectronics

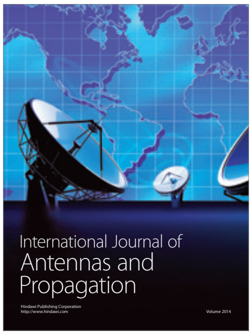
VLSI Design

International Journal of
Navigation and
Observation

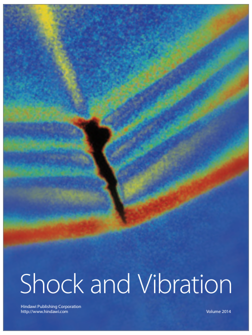

Modelling &
Simulation
in Engineering

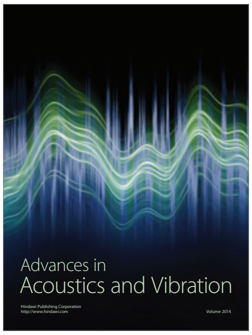
International Journal of
Aerospace
Engineering


International Journal of
Chemical Engineering


International Journal of
Antennas and
Propagation

Active and Passive
Electronic Components


Shock and Vibration


Advances in
Acoustics and Vibration