Research Article

A Middleware Approach to Achieving Fault Tolerance of Kahn Process Networks on Networks on Chips

Onur Derin,¹ Erkan Diken,² and Leandro Fiorin¹

¹ *ALaRI, Faculty of Informatics, University of Lugano, 6900 Lugano, Switzerland*

² *Faculty of Electrical Engineering, Eindhoven University of Technology, 5600 MB Eindhoven, The Netherlands*

Correspondence should be addressed to Onur Derin, derino@alari.ch

Received 28 August 2010; Revised 12 January 2011; Accepted 24 March 2011

Academic Editor: Michael Hübner

Copyright © 2011 Onur Derin et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Kahn process networks (KPNs) is a distributed model of computation used for describing systems where streams of data are transformed by processes executing in sequence or parallel. Autonomous processes communicate through unbounded FIFO channels in absence of a global scheduler. In this work, we propose a task-aware middleware concept that allows adaptivity in KPN implemented over a Network on Chip (NoC). We also list our ideas on the development of a simulation platform as an initial step towards creating fault tolerance strategies for KPNs applications running on NoCs. In doing that, we extend our SACRE (Self-Adaptive Component Run Time Environment) framework by integrating it with an open source NoC simulator, Noxim. We evaluate the overhead that the middleware brings to the total execution time and to the total amount of data transferred in the NoC. With this work, we also provide a methodology that can help in identifying the requirements and implementing fault tolerance and adaptivity support on real platforms.

1. Introduction

Past decade has witnessed a change in the design of powerful processors. It has been realized that running processors at higher and higher frequencies is not sustainable due to unproportional increases in power consumption. This led to the design of multicore chips, usually consisting of multiprocessor symmetric Systems on Chip (MPSoCs), with limited numbers of CPU-L1 cache nodes interconnected by simple bus connections and capable in turn of becoming nodes in larger multiprocessors. However, as the number of components in these systems increases, communication becomes a bottleneck and it hinders the predictability of the metrics of the final system. Networks on Chip (NoCs) [1] emerged as a new communication paradigm to address scalability issues of MPSoCs. Still, achieving goals such as easy parallel programming, good load balancing and ultimate performances, dependability and low-power consumption pose new challenges for such architectures. Technology scaling decreases the yield in manufacturing and thermal effects cause defects at run time. Thus fault tolerance

becomes a major concern not only for economical reasons but also for the end user.

In addressing these issues, we adopted a component-based approach based on Kahn Process Networks (KPNs) for specifying the applications [2]. KPN is a stream-oriented model of computation based on the idea of organizing an application into streams and computational blocks; streams represent the flow of data, while computational blocks represent operations on a stream of data. KPN presents itself as an acceptable tradeoff point between abstraction level and efficiency versus flexibility and generality. It is capable of representing many signal and media processing applications, which occupy the largest percentage of the consumer electronics in the market.

Our eventual goal is to run a KPN application directly on an NoC platform with self-adaptivity and fault-tolerance features. It requires us to implement a KPN run-time environment that will run on the NoC platform and support adaptation and fault-tolerance mechanisms for KPN applications on such platforms. In order to achieve our goal, we propose the use of a self-adaptive run-time environment

(RTE) that is distributed among the tiles of the NoC platform. It consists of a middleware that provides standard interfaces to the application components allowing them to communicate without knowing about the particularities of the network interface and the communication network in general. Moreover, the distributed run-time environment can manage the adaptation of the application for high-level goals such as fault-tolerance, high performance, and low-power consumption by migrating the application components between the available resources and/or increasing the parallelism of the application by instantiating multiple copies of the same component on different resources [3, 4]. Such a selfadaptive RTE constitutes a fundamental part in order to enable systemwide self-adaptivity and continuity of service support [5].

In view of the goal stated above, we propose to use our SACRE framework [4] that allows creating self-adaptive KPN applications. In [3], we listed platform level adaptations and proposed a middleware-based solution to support such adaptations. In the present paper, we define the details of the self-adaptive middleware particularly for NoC platforms. In doing that, we choose to integrate SACRE with the Noxim NoC simulator [6] in order to realize functional simulations of KPN applications on NoC platforms.

An important issue regarding the NoC platform is the choice of the communication model. Depending on the NoC platform, we may have a shared memory space with the Non-Uniform Memory Access (NUMA) model or we may rely on pure message passing with the No Remote Memory Access (NORMA) model [7]. While NUMA case, the implementation of the KPN semantics is straightforward as long as the platform provides some synchronization primitives, on the NORMA case it represents the main challenge.

The remainder of this paper is organized as follows. Section 2 overviews related work. Section 3 explains details of the middleware in the NORMA case. Section 4 presents our implementation of the middleware by integrating SACRE and Noxim. Section 5 discusses an analytical model for the calculation of the communication traffic and computational time for a given application, while Section 6 presents evaluation results for a JPEG case study. Section 7 provides the requirements for the implementation of fault tolerance mechanisms and lists some application level fault tolerance patterns. Section 8 presents conclusions and future work.

2. Related Work

Kahn process networks (KPNs) is a widely studied distributed model of computation used for describing systems where streams of data are transformed by processes executing in sequence or parallel [2]. Previous research on the use of KPN in multiprocessor embedded devices have been mainly focusing on the design of frameworks which employ them as model for the application [8–10], and which aim at supporting and optimizing the mapping of KPN processes on the nodes of a reference platform [11, 12]. In [8, 9], different methods and tools are proposed for automatically generating

KPN application models from programs written in Matlab or 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 [10, 12], or architecture synthesis for FPGA platforms [8] is implemented.

Software synthesis relies on the high-level APIs provided by the reference platform for facilitating the programming of a multiprocessor system. The trend from single-core design to many-core design has forced to consider interprocessor communication issues for passing the data between the cores. One of the emerged message passing communication API is Multicore Association's Communication API (MCAPI) [13] that targets the intercore 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 [14].

However, the communication primitives available with these message passing libraries do not support the blocking write operation as required by KPN semantics. Main features in order to implement KPN semantics are blocking read and, in the limited memory case, blocking write. Key challenge is the implementation of the blocking write feature. There are different approaches addressing this issue. In [15], a programming model is proposed based on the MPI communication primitives (`MPI_send()` and `MPI_recv()`). `MPI_recv()` blocks the task until the data is available while `MPI_send()` is blocking until the buffer is available on the sender side. Blocking write feature is implemented via operating system communication primitives that ensure the remote processor buffer has enough space before sending the message. Another approach is presented in [16], a network end-to-end control policy is proposed to implement the blocking write feature of the FIFO queues.

In this work, we propose an active middleware layer that implements the blocking write feature through *virtual connectors*, that are introduced in opposite directions to the original ones. This work introduces an approach which is based on a novel implementation of KPN semantics which can be employed on Network-on-Chip platforms. The use of this approach can be considered complementary to the aforementioned related work about generation and mapping of KPN processes, being in fact one of our goals the proposal and evaluation of a middleware that can be employed by these types of tool as a support for the programmability of the cores, when requesting adaptability of the applications.

Similarly to [16], we target as reference platform a NoC architecture. However, with respect to [16], instead of directly relying on the hardware NoC buffers for implementing the KPN FIFOs, we decided to maintain them implemented in software, in order to keep the flexibility needed in an adaptive application.

The *virtual connector* concept for implementing the blocking write feature can be considered in part similar to other general protocols in which data are pulled rather than pushed. An example can be, for instance, found in [17], in the case of the implementation of node-to-node communication in asynchronous NoCs. However, the work

in [17] mainly addresses the aspects of design and simulation of asynchronous NoC platforms, while in our case this type of protocol has been introduced, for the first time, as a modification of our middleware to support blocking writes in KPN, independently of the implementation of the hardware NoC platform below.

In this work, we also deal with the aspects of fault tolerance of KPNs. Fault tolerance has been object of a large amount of research, addressing both the hardware and software aspects of a platform [18]. When focusing on software techniques, several approaches were proposed, suggesting for instance the use of N-version software [19], self-checking software [20], Multiple-Task N-Modular Redundancy [21], or, more recently, the use of data and code duplications for detecting and correcting transient faults affecting the processor data segment and control flow duplication for correcting and detecting faults in the code segment [22, 23]. In KPNs, reconfiguration algorithms for dealing with single or multiple faulty nodes or channels were presented in [24, 25]. Algorithms are based on the redistribution of the faulty nodes actions to other nodes, as well as the input and output channels of the faulty nodes. A set of rules is presented that should be followed at the occurrence of faults in order to minimize data loss. The approach presented in this paper work can be considered complementary to the one presented in [24, 25]. Differently from [24, 25], this work presents in fact the implementation of fault tolerant techniques in KPNs that allow the detection and masking of faults in hardware elements running KPN processes and evaluates the associated costs. Reconfiguration and task remapping would be a consequence of the detection of a faulty hardware component.

Another fundamental contribution of our approach is moreover the separation of fault tolerance concerns from the functional application development. The proposed fault tolerance techniques can be applied at run time without manual intervention of the application developer.

3. Task-Aware Middleware

A KPN application consists of a set of parallel running tasks (application components) connected through nonblocking write and blocking read unbounded FIFO queues (connectors) that carry tokens between input and output ports of application components. A token is a typed message. Figure 1 shows a simple KPN application. Running a KPN application on an NoC platform would require to map the application components on the several tiles of the NoC platform.

Given the assumption that we have a heterogeneous NoC-based platform and that the application will be specified using the KPN model of computation, we need a middleware as a software layer that runs on the programmable cores and as a hardware wrapper for nonprogrammable cores. The main functionality of the middleware is to implement KPN semantics for the application tasks that are either software or hardware tasks. However, we put some more requirements due to the fault-tolerance goal. In this section the middleware requirements is discussed and a middleware solution is

FIGURE 1: A simple KPN application with application components A, B, C, D, E, and F.

described for programmable and nonprogrammable cores that addresses those requirements.

3.1. Requirements. When deciding on the implementation details of a middleware that will support running of a KPN application on a NoC platform, we came up with some requirements for the middleware. The most fundamental requirement for a middleware to support KPN semantics is the ability to transfer tokens among tiles assuring blocking read. Since unbounded FIFOs cannot be realized on real platforms, FIFOs have to be bounded. Parks' algorithm [26] provides bounds on the queue sizes through static analysis of the application. In the case of bounded queues, blocking write is also required to be supported.

Another requirement is that we would like to have platform-independent application components. This will make it easier to program for the platform by allowing the development of application components in isolation and running them without modifications. This can be achieved by separating the KPN library that will be used to program the application from the communication primitives of the platform. Middleware will link the KPN library to the platform specific communication issues.

In line with the above requirement, we would like that application components are not aware of the mapping of components on the platform. They should only be concerned with communicating tokens to certain connectors. Therefore the middleware should enable mapping-independent token routing. These requirements are of great importance if we want to achieve fault tolerance and adaptivity of KPN applications on NoC platforms in such a way that assures separation of concerns. This means that it is the platform that provides fault tolerance and adaptivity features to the application and not the application developer.

3.2. Middleware Implementation in the NORMA Case. In the NORMA model, tasks only have access to the local memory and there is no shared address space. Therefore tasks on different tiles have to pass the tokens among each other via message passing routines supported by the network interface (NI).

In order to address the middleware requirements previously listed, our key argument is the implementation of an active middleware layer that appears as a KPN task and gets connected to other application tasks with the connectors of

FIGURE 2: KPN example mapped on 2×2 mesh NoC platform.

the specific KPN library that is adopted by the application components. Opposedly, a passive middleware layer would be a library with a platform specific API for tasks to receive and send tokens.

We build our middleware on top of *MPI_recv()* and *MPI_send()* primitives. These methods allow sending/receiving data to/from a specific process regardless of which tile the process resides on. *MPI_recv()* blocks the process until the message is received from the specified process-tag pair. *MPI_send()* is nonblocking unless there is another process on the same tile that has already issued an *MPI_send()*. *MPI_send()* also blocks when the remote buffer is full.

Every tile has a middleware layer that consists of middleware sender and receiver processes. Figure 2 shows the middleware layers and a possible mapping of the example pipeline on four tiles of a 2×2 mesh NoC platform. There is a sender process for each outgoing connector. An outgoing connector is one that is connected to an input port of the application component that resides on a different tile. Similarly, there is a receiver process for each incoming connector. These processes are actually KPN tasks with a single port. This is an input port for a sender process and an output port for a receiver process. The job of a sender middleware task is to transmit the tokens from its input over the network to the receiver middleware task on the corresponding tile (i.e., the tile containing the application component to receive the token). Similarly, a receiver middleware task should receive the tokens from the network and put them in the corresponding queue. Figure 3 shows the sender and receiver middleware tasks between the ports of application components B and C.

We need to implement a blocking write blocking-read bounded channel that has its source in one processor and its sink in another one. *MPI_send()* as described above does

FIGURE 3: Middleware details for the connector between B and C.

Loop

```

t ← input_port.read()
MPI_recv(tvc, PR, vc)
MPI_send(t, PR, oc)
end loop

```

ALGORITHM 1: Sender middleware task per outgoing connector (t : data token, t_{vc} : dummy token for virtual connector, P_R : process identifier of the remote middleware task, vc : virtual connector tag, oc : original connector tag).

not implement blocking write operation. It can be modified and be implemented in such a way that it checks whether the remote queue is full or not by using low-level support from the platform [15, 16]. In order to do this in a way that would not require changes to the platform, we make use of the *virtual connector* concept. A virtual connector is a software queue that is connected in the reverse direction to the original connector. For every connector between sender and receiver middleware tasks, we add a virtual connector that connects the receiver middleware task to the sender middleware task. Figure 3 shows the virtual connector along with the sender and receiver middleware tasks for the outgoing connector from application component B to C. The receiver task initially puts as many tokens to the virtual connector as the predetermined bound of the original connector. The sender has to read a token from the virtual connector before every write to the original connector. Similarly, the receiver has to write a token to the virtual connector after every read from the original connector. Effectively, the virtual connector enables the sender to never get blocked on a write. The read/write operations from/to original and virtual connectors can thus be done using *MPI_send()* and *MPI_recv()* as there is no more need for blocking write in presence of virtual connectors. Algorithms 1 and 2 show the pseudocodes for sender and receiver middleware tasks, respectively.

With the middleware layer, an outgoing blocking queue of bound b in the original KPN graph is converted into three blocking queues: one with bound $b1$ between the output


```

for  $i = 1$  to connector_bound do
   $MPI\_send(t_{vc}, P_R, vc)$ 
   $i \leftarrow i + 1$ 
end for
Loop
   $MPI\_recv(t, P_R, oc)$ 
   $MPI\_send(t_{vc}, P_R, vc)$ 
   $output\_port.write(t)$ 
end loop

```

ALGORITHM 2: Receiver middleware task per incoming connector.

port of the source component and the sender middleware task; one with bound b_2 between the sender and receiver middleware tasks; one again with bound b_2 between the receiver middleware task and the input port of the sink component. Values b_1 and b_2 can be chosen freely such that $b_1 + b_2 \geq b$ and $b_1, b_2 > 0$.

If the middleware layer is not implemented as an active layer, then the application tasks would need to be modified to include the virtual connectors. Moreover, use of virtual connectors enables us to not require changes to the NoC for custom signalling mechanisms.

Another benefit of having virtual connectors is in avoiding deadlocks. Since $MPI_send()$ can be issued by different middleware tasks residing on the same tile in a mutually exclusive way, there may be deadlock situations for some of the task mapping decisions. For example, consider the case (see Figure 1 and the mapping in Figure 2) where an application task (C) is blocked on a call to $MPI_send()$ until the queue on the receiver end is not full. It may be that the application task on the receiver end (E) is also blocked waiting for a token from an application task (D) on the tile where C resides. Since tasks on the same tile has to wait until the $MPI_send()$ call of the other task returns, D cannot write the token to be received by E. Therefore we have a deadlock situation where C is blocked on E, E is blocked on D, and D is blocked on C. With virtual connectors, it is guaranteed that an $MPI_send()$ call will not ever be blocked.

The problem of deadlocking can be solved also without using virtual connectors. However, that would require implementing expensive distributed conditional signalling mechanisms on the NoC or inefficient polling mechanisms.

3.3. KPN Middleware Wrapper for Nonprogrammable Cores. For the case of nonprogrammable cores, the middleware can be implemented as a hardware KPN wrapper. It will allow treating those cores as KPN tasks. This wrapper will include input and output FIFO buffers and KPN controller logic as shown in Figure 4. These additional units will reside along with the hardware core and the network interface (NI) in the tile. The functionality of the middleware as described in the previous section will be implemented by the KPN controller logic. It will initially send as many virtual tokens as the predetermined size of the input buffer. Whenever there

FIGURE 4: Block diagram of the KPN middleware for a nonprogrammable core (N-P Core).

is a new packet coming in, the NI will pass the received packet to the KPN controller which will extract the token to be put into the input buffer and signal this event to the KPN controller. In response, the controller will send back a virtual token and it will activate the hardware core to execute using the first token in the input FIFO. After the completion of the execution, the controller will wait until it receives a virtual token before writing the result to the output buffer. The KPN controller will send the tokens in the output FIFO by wrapping them into middleware packets by adding their destination component information and passing them over to the NI. The KPN controller may possibly interact with another controller that implements the self-checking and/or self-testing policies or its functionality may be extended for those purposes. Thus the KPN controller, when feeding nominal input to the core has to work in accordance with the specific interface of the nonprogrammable core.

4. Middleware Implementation

In order to realize functional simulation of the proposed system, and testing its functionalities, we integrated our SACRE tool with an NoC simulator.

Noxim [6] is an open source and cycle-accurate simulator developed in SystemC. It can generate NoC models by setting several configuration parameters such as the network size, the routing algorithm, and the traffic type. Models generated implement wormhole control flow, in which NoC packets are divided in an arbitrary number of flits which all follow the route taken by the first one which contains the header of the packet and which specifies the destination address. The NoC models generated by Noxim allow to analyze and evaluate a set of quality indices such as delay, throughput, and energy consumption.

SACRE [4] is a component framework that allows creating self-adaptive applications based on software components and incorporates the Monitor-Controller-Adapter loop [5] with the application pipeline. The component model is based on KPN. It supports run-time adaptation of the pipeline via parametric and structural adaptations.

4.1. Integration of SACRE and Noxim. The integration of SACRE and Noxim is done in order to be able to simulate

TABLE 1: Port connection table for Figure 1.

Source		Destination	
component	port	component	port
A	out1	B	in1
B	out1	C	in1
B	out2	D	in1
C	out1	E	in1
D	out1	E	in2
E	out1	F	in1

TABLE 2: Component mapping table for Figure 2.

Component	Tile
A	0
B	0
C	1
D	1
E	2
F	3

KPN applications on NoC platforms. The proposed middleware is implemented for the NORMA case. First of all, we do not have the `MPI_send()` and `MPI_rcv()` primitives in Noxim. Actually the simulator does not even implement a proper network interface, while it provides traffic generators and sinks. We implemented the transport layer such that we can send data and reconstruct the data on the other end. In absence of MPI primitives in SACRE-Noxim, we implemented the task-aware middleware over the transport layer of the NoC network interface as described below.

We conceived the middleware as a KPN task by extending it from *SACREComponent* in order to be able to connect it to the queues of the local application tasks. In the SACRE framework, a *SACREComponent* is the base class for all the KPN tasks and allows them to specify their input/output ports and functions [4]. To have a complete communication scheme, two types of middleware tasks are implemented, *MWSender* and *MWReceiver*.

Implementation details of the SACRE and Noxim integration is depicted in Figure 5. We will refer to this scheme in order to explain the details of the integration. It is assumed that two application components are mapped on two different tiles of the NoC. One of the application components, which is mapped to tile 0, produces tokens, while the other application component, which is accommodated on tile 1, consumes token.

On the sender side, since middleware tasks and application components are SACRE components and are connected with blocking queues, the producer application component writes tokens into the queue that resides between producer component and *MWSender* task. Blocking queues block the SACRE components in case of an attempt to read from an empty queue or an attempt to write into a full queue. *MWSender* uses `MPI_send()` primitive in order to send tokens to the destination application component after reading tokens from its corresponding input port. `MPI_send()`

has the signature shown in Algorithm 3. As *MWSender* task is independent from the application components, it accepts generic data types. Furthermore, a *port connection table* is accessed so as to get the destination component name and destination port identifier which are passed as arguments to the `MPI_send()` call. As shown in Table 1, the port connection table represents the KPN application and shows which components and which ports are connected with each other.

When there is a token to be forwarded, `MPI_send()` is called. `MPI_send()` wraps the token into a *MWPacket* object as shown in Figure 6 by adding the destination task name and destination port name as the header information. Since, only information related to destination is passed as parameters to the `MPI_send()`, in addition to these information, *port connection table* is accessed in order to add source component name and source port name to the *MWPacket*.

In the following step, the *MWPacket* object is sent via the NI to the destination tile by wrapping it in a *NIPacket* object. *NIPacket* has the structure shown in Figure 6. The destination tile identifier is looked up from the *component mapping table*. This table stores which components reside on each tile as shown in Table 2. Currently NI transfers packets splitting them into several flits through wormhole routing.

On the receiver side, the network interface receives the flits and transport layer reconstructs the *NIPacket* object. Then the receiver process extracts the *MWPacket* from *NIPacket* and puts it in the *receive list*. This receive list is accessed by `MPI_rcv()` primitive which is called by *MWReceiver* tasks. `MPI_rcv()` has the signature shown in Algorithm 3.

MWReceiver tasks are connected to the application components which have input ports for receiving tokens. Similarly to the *MWSender*, output ports of the *MWReceiver* tasks are connected to the input ports of the application components through blocking queues. Differently from *MWSender* tasks, *MWReceiver* tasks use `MPI_rcv()` primitive in order to read tokens from the receive lists of the corresponding processing element. *Port connection table* is accessed to get the source component name and source port which are used to check the receive list in order to identify the received packet. Then, token is passed to the application component by writing the token into the blocking queue via corresponding port.

4.2. Case Study: Simulation of JPEG Encoder and Decoder.

In order to verify the correctness of the integration of SACRE and Noxim, we run a KPN-modeled application on the Noxim simulator. Components of the application were mapped on different tiles of NoC. In order to do that, as a first step, a JPEG Encoder & Decoder application is implemented by using the SACRE KPN library. This allows to verify the functional correctness of the application before running it on the NoC platform. Figure 7 depicts the application components of the JPEG Encoder & Decoder.

Second step requires mapping the application components on the tiles of NoC. Figure 8 shows one possible mapping of the JPEG application components on different

FIGURE 5: Implementation details of the SACRE and Noxim Integration.

```

MPI_rcv(recv_buffer, size, src, tag)
MPI_send(send_buffer, size, dest, tag)

```

ALGORITHM 3: Signatures of `MPI_rcv()` and `MPI_send()` (`recv_buffer`: memory to allocate received token, `send_buffer`: allocated memory for token to be sent, `size`: size of the data token, `src`: source component, `dest`: destination component, and `tag`: port identifier).

tiles of the 4×4 NoC. Application is mapped such that all communicating components have a Manhattan distance of one. It is also possible to map more than one component on one tile of NoC. The yellow arcs refer to the data flow between components.

Application mapping is accomplished by simply inserting component and tile information into the *component mapping table*. Additionally, it is required to insert information about the connection of the components into the *port connection table* by providing input and output port identifiers with component names. The next step requires to create the application components and to connect them with their related middleware tasks. It is important to note that bound of the blocking queues that reside between components and middleware tasks is provided during that connection phase. Similar to the bounds on blocking queues, number of the virtual tokens are decided when each *MWReceiver* task is created.

Functional correctness is tested by giving a gray scale image as an input, after encoding and decoding, an output image is produced as an output at the end of the simulation. Moreover, as stated previously, Noxim simulator provides

statistics in order to be able to evaluate the interconnection network traffic.

5. Middleware Analytical Models

In this Section, we present and discuss an analytical model for calculating the communication load and the execution time associated with an application, in order to be able to evaluate the proposed middleware in terms of the overhead it introduces. In particular, as shown in Section 6, we focus our evaluation on the JPEG case study.

The following analytical model is proposed.

- (i) A *KPN task graph* $g_t = (V_t, E_t)$ is composed of tasks $t \in V_t$ and data dependencies (connectors) $e \in E_t \subseteq V_t \times V_t$.
- (ii) An *architecture graph* $g_a = (V_a, E_a)$ is composed of processing nodes $n \in V_a$ and bidirectional communication links $l \in E_a \subseteq V_a \times V_a$.
- (iii) A *task mapping* $\beta_t : V_t \rightarrow V_a$ is an assignment of tasks $t \in V_t$ to nodes $n \in V_a$.
- (iv) A *communication mapping* $\beta_c : E_t \rightarrow E_a^i$ is an assignment of data dependencies $e \in E_t$ to paths of length i in the architecture graph g_a . A *path* p of length i is given by i -tuple $p = (l_1, l_2, \dots, l_i)$.
- (v) *path* $(V_a, V_a) \rightarrow E_a^i$ is a function that implements a deterministic routing algorithm and returns a path between two given nodes. *Path set* P is the set of paths between all node pairs:

$$P = \{p_k : p_k = \text{path}(n_i, n_j), \forall n_i, n_j \in V_a \wedge n_i \neq n_j\}. \quad (1)$$

FIGURE 6: Structure of packets at the NI and middleware levels (src_id: source tile, dst_id: destination tile, ts: timestamp, dst_comp: destination component, dst_port: destination port, src_comp: source component, and src_port: source port).

FIGURE 7: JPEG Encoder & Decoder implemented with SACRE KPN library.

FIGURE 8: Mapping of JPEG application components on 3x3 NoC.

Initial and final nodes of a path can be obtained by *source* and *sink* functions

$$p_k = \text{path}(n_i, n_j) \Rightarrow \text{source}(p_k) = n_i \wedge \text{sink}(p_k) = n_j. \quad (2)$$

- (vi) The task graph can be annotated with demand values where *demand* d_i on a data dependency $e_i \in E_t$, denotes the amount of data transferred between the two tasks. Demand values are obtained considering the network protocol of the platform spanning all layers from application and middleware to the network interface. At the application level, the number of writes w_i for each connector e_i is calculated by profiling the application with a test input. Let s_i

be the size (in bytes) of the data type of the data transferred on connector e_i . Then total number of bytes transferred at the application level on connector e_i is

$$d_i^{\text{app}} = w_i \times s_i. \quad (3)$$

At the middleware level, each token of the application is augmented with the MW header data (see Figure 6). Let h_{mw} be the size of the header section of the *MWPacket*. The amount of data to be supplied to the network interface by the middleware is

$$d_i^{mw} = w_i \times (s_i + h_{mw}). \quad (4)$$

Finally, at the NI level, *NIPackets* are created by adding the header information (see Figure 6) to the *MWPacket* and transfer the packet in flits of size f bytes. Let h_{ni} be the size of the header section of the *NIPacket*. Then the total number of bytes transferred at the NI level for connector e_i is

$$d_i^{ni} = w_i \times \left(\left(\left\lceil \frac{s_i + h_{mw}}{f} \right\rceil \times f \right) + h_{ni} \right). \quad (5)$$

Due to the virtual connector that is connected in the reverse direction, there is an additional transfer of a virtual token for every nominal token. We represent the virtual token as a single header flit in the network, thus incurring an additional traffic of $w_i \times h_{ni}$. The number of bytes transferred at the NI level for connector e_i with our middleware is

$$d_i = w_i \times \left(\left(\left\lceil \frac{s_i + h_{mw}}{f} \right\rceil \times f \right) + 2 \times h_{ni} \right). \quad (6)$$

If the task mapping is fixed and no task will be migrated at run time, tasks can be programmed by embedding the mapping information in them in which case the middleware would not be needed. In such a case, the number of bytes transferred for connector e_i without a middleware is

$$d_i = w_i \times \left(\left(\left\lceil \frac{s_i}{f} \right\rceil \times f \right) + h_{ni} \right). \quad (7)$$

- (vii) Core type set C consists of core types C_i and lists the types of cores available in a given NoC platform.

5.1. Communication Cost. In order to formulate the analytical model for the calculation of total amount of data transferred in the network during the execution of the

application, we define several application-specific parameters, architecture-specific parameters, and mapping-specific variables in the form of incidence matrices.

X^{NT} is an incidence matrix of size $|V_a| \times |V_t|$ that denotes the mapping of tasks onto the nodes.

$$X_{ij}^{NT} = \begin{cases} 1 & \text{if } t_j \in V_t \text{ is bound onto node } n_i \in V_a, \\ 0, & \text{otherwise.} \end{cases} \quad (8)$$

Y^{PE} is an incidence matrix of size $|P| \times |E_t|$ that denotes which path realizes which data dependency

$$Y_{ij}^{PE} = \begin{cases} 1 & \text{if } e_j \in E_t \text{ is mapped to } p_i \in P, \\ 0, & \text{otherwise.} \end{cases} \quad (9)$$

M^{TE} is an oriented incidence matrix of size $|V_t| \times |E_t|$ that relates the tasks to the data dependencies. For a given task graph, M^{TE} is known

$$M_{ij}^{TE} = \begin{cases} 1 & \text{if } \exists t_k, e_j = (t_i, t_k) \in E_t, \\ -1 & \text{if } \exists t_k, e_j = (t_k, t_i) \in E_t, \\ 0, & \text{otherwise.} \end{cases} \quad (10)$$

M^{PL} is an incidence matrix of size $|P| \times |E_a|$ that denotes the relation between all paths resulting from a given deterministic routing algorithm and the links that make up the path. For a given routing algorithm and architecture graph, M^{PL} is known.

$$M_{ij}^{PL} = \begin{cases} 1 & \text{if } l_j \in p_i, \\ 0, & \text{otherwise.} \end{cases} \quad (11)$$

Total Amount of Data Transferred. The total amount of data transfer, d_{tot} , on the links can be calculated as the sum of all demands d_i on the links of the paths that arise according to a given mapping with the following equation:

$$d_{\text{tot}} = d^T Y^{EP} M^{PL} \mathbf{1}_{|E_a|}, \quad (12)$$

where $\mathbf{1}_m$ is a matrix of size $m \times 1$ with all elements equal to 1. It is to be noted that $Y^{EP} = (Y^{PE})^T$. Similar relation holds for all defined matrices.

This static model for communication has also been used in [27] and disregards the congestion on the links. However at low load conditions, it is argued that it is a good approximation.

Note that the communication cost takes into account the intertile communication done over the NoC between tasks and not the intratile communication when communicating tasks are mapped onto the same node.

5.2. Computational Cost. In order to formulate the model for calculating total execution time of the application, we define additional parameters in matrix form, namely, M_{cap}^{TC} , T_{cap}^{TC} , and M^{NC} .

M_{cap}^{TC} is an incidence matrix of size $|V_t| \times |C|$ that denotes which core types are capable of realizing which tasks. Programmable cores would be capable of realizing different kinds of task functionalities, whereas nonprogrammable cores would have dedicated functions

$$M_{\text{cap } ij}^{TC} = \begin{cases} 1 & \text{if } C_j \in C \text{ is capable of realizing task } t_i \in V_t, \\ 0, & \text{otherwise.} \end{cases} \quad (13)$$

T_{cap}^{TC} is a matrix of size $|V_t| \times |C|$ that denotes the completion times of all tasks on all core types for a test input. Given an application and architecture, this matrix can be obtained by offline profiling

$$T_{\text{cap } ij}^{TC} = \begin{cases} \text{completion time of } t_i \text{ on } C_j & \text{if } M_{\text{cap } ij}^{TC} = 1, \\ 0 & \text{if } M_{\text{cap } ij}^{TC} = 0. \end{cases} \quad (14)$$

M^{NC} is an incidence matrix of size $|V_a| \times |C|$ that denotes the core type of the architectural nodes. Given an architecture, M^{NC} is known

$$M_{ij}^{NC} = \begin{cases} 1 & \text{if } n_i \in V_a \text{ is of core type } C_j \in C, \\ 0 & \text{otherwise.} \end{cases} \quad (15)$$

M^{TC} is an incidence matrix of size $|V_t| \times |C|$ that denotes the actual mapping of tasks on core types. Given a task to node mapping matrix, X^{TN} , it can be calculated as

$$M^{TC} = X^{TN} M^{NC}. \quad (16)$$

T^T is a matrix of size $|V_t| \times 1$ that denotes the completion time of the task on the core type that it is mapped onto. It can be calculated as

$$T^T = (M^{TC} \cdot T_{\text{cap}}^{TC}) \mathbf{1}_{|C|}, \quad (17)$$

where the “ $\cdot$ ” operator represents elementwise matrix multiplication.

T^N is a vector of size $|V_a| \times 1$. T_i^N denotes the sum of execution times of tasks that are mapped on the same node, n_i . It can be calculated as

$$T^N = X^{NT} T^T. \quad (18)$$

Total Execution Time of Tasks. We calculate the total computation time of tasks by finding the maximum of the sum of the execution times of tasks mapped on the same core.

$$\max(T^N) = \max(X^{NT} ((X^{TN} M^{NC}) \cdot T_{\text{cap}}^{TC}) \mathbf{1}_{|C|}), \quad (19)$$

where $\max$ is a function that returns the maximum value in a given vector.

This is a static model that has been also used in [28] and disregards the context switching times. This model can be considered to have a reasonable accuracy in the case of

typical streaming applications when there is enough inter- and intra-process parallelism such that the resources do not stall because of blocking and when communication is smooth.

However, it is of particular importance in our case to incorporate the context switching times because the proposed middleware tasks work as threads. In order to calculate the number of context switches on a core, we adopt a scheduler that executes a task until it is blocked on a read or a write. In the particular case of m pipelined tasks (t_1 to t_m) with single input and output connectors mapped on the same node n_i where t_j is connected to t_{j+1} through the bounded connector (t_j, t_{j+1}) with equal sized bounds b , the number of context switches in node n_i can be calculated as

$$N_{CSi} = m \times \left\lceil \frac{w}{b} \right\rceil - 1, \quad (20)$$

where w is the number of writes to a connector and assumed to be equal for all connectors.

For the particular mapping of the JPEG application in our case study, this formula will suffice because the mapping conforms to the assumptions of the above analysis (see Figure 8). We leave it as an open issue to calculate the number of context switches for general mappings and schedulers.

The total time T_{CSi} , spent in making N_{CSi} context switches can be calculated by

$$T_{CSi} = N_{CSi} CS_{cap_i}, \quad (21)$$

where CS_{cap_i} is the single context switching time of the core type of node n_i .

With the inclusion of context switching time, the total execution time of the application is

$$T^{app} = \max(T^N + T_{CS}). \quad (22)$$

6. Evaluation of Results

In this section, we evaluate the overhead associated with the use of the middleware in the case of the JPEG case study. Being our work a first attempt to evaluate the cost of using a middleware layer for supporting the adaptivity of the application, we cannot directly compare our results with those obtained in previous work adopting the KPN model of computation. We therefore present and discuss the costs of adding this additional layer. We calculate the computation and communication overhead by means of the analytical model presented in Section 5. The computational overhead is interpreted as the increase in the total execution time of an application. In the case of our case study, we look at the increase in the processing time of a single image. The communicational overhead is interpreted as the increase in the communication cost which is defined as the total amount of data transferred in the interconnection network.

6.1. Communication Overhead. In the case of our middleware, allocating 1 byte for each of *dst_comp*, *dst_port*, *src_comp*, *src_port* headers and 4 bytes for the *size* header, we have $h_{mw} = 8$. In our model of the NoC, we consider a

datawidth of the physical link of 32 bits. We assume an XY-routing algorithm, and a wormhole control flow, in which each packet is divided in an arbitrary number of flits, whose dimension is equal to the width of the physical link. The flits of a packet follow the route taken by the first one, into which routing information are inserted (header). Our network interface has therefore an *NIPacket* header of size 1 flit, thus $h_{ni} = f = 4$ bytes. In the JPEG application, all the connectors are either of data type *Token<int>* or *Token<float>* which are both 8 bytes giving $s_i = 8$ bytes for all connectors e_i .

Evaluating (6), (7), and (12) for the mapping of JPEG application shown in Figure 8, we obtain a communication overhead of 1.00, meaning double amount of data transfer in the network, when compared to the transmission of the information without the use of the middleware. As the size of the data type of tokens transferred on the connectors increases, the overhead due to the middleware decreases. In that respect, the JPEG application with only 8-byte-long tokens gives a higher communication overhead.

Figure 10(a) shows the overhead for varying token sizes (in bytes) in case these would be the token sizes in the JPEG case study. It can be seen that the communication overhead falls under 10% when the tokens are bigger than 116 bytes. It should be noted that token size is an application-dependent parameter, and depending on the application, the application programmer can modify the application to work with bigger token sizes with additional effort. Although increasing token size results in lower communication overhead, due to the pipelined propagation of wormhole switching the communication could not be considered as ideal anymore, and delay due to the possible contention should be taken into account in the choice of the right token dimension [29]. Another possibility could imply the propagation of the same token by using several network packets, therefore taking into consideration in the calculation of the communication overhead the traffic generated by the header flits of the additional packets employed.

6.2. Computational Overhead. We profiled the JPEG encoder/decoder application by using the SESC simulator [30] with a test input image of 211×205 pixels and by considering a MIPS processor running at 5 GHz. As done for the calculation of the communication overhead, we consider tokens of 8 bytes. The execution times of the application tasks are shown in Table 3. The execution times of the middleware tasks for each connector are shown in Table 4. In the tables, values of execution time that are equal to 0.00 are due to the rounding to the second decimal number. As can be noticed, the execution time of the middleware tasks increases with the amount of data they transfer. The number of writes to the connectors are shown in Table 5. We used the mapping and the XY-routing-based NoC platform shown in Figure 8 and a context switching time of $0.16 \mu s$ for the processor.

Using (22) and varying the bound of the connectors, we obtained the overhead results shown in Figure 10(b). The results show that for $b > 24$, the computational overhead can

FIGURE 9: The distribution of total execution time between application tasks, middleware tasks, and context switching for the JPEG encoder/decoder case study.

FIGURE 10: The communication and computational overhead due to the proposed middleware for the JPEG encoder/decoder case study.

TABLE 3: Execution times (in *ms*) of tasks on the available core types (T_{cap}^{CT}).

Core type	Tasks												
	src t_1	fork t_2	r2b t_3	dct t_4	quan t_5	zzenc t_6	rlenc t_7	rldec t_8	zzdec t_9	norm t_{10}	idct t_{11}	b2r t_{12}	sink t_{13}
MIPS	13.89	0.00	1.84	6.43	1.81	2.01	1.35	1.12	1.95	1.81	9.08	1.81	8.75

TABLE 4: Execution times (in *ms*) of MWSender and MWReceiver tasks for each connector.

Core type	src → fork $e_{(1,2)}$	fork → r2b $e_{(2,3)}$	fork → b2r $e_{(2,12)}$	fork → sink $e_{(2,13)}$	src → r2b $e_{(1,3)}$	dct → quan $e_{(4,5)}$	rlenc → rldec $e_{(7,8)}$	norm → idct $e_{(10,11)}$	idct → b2r $e_{(11,12)}$	b2r → sink $e_{(12,13)}$
MIPS	0.00	0.00	0.00	0.00	2.24	2.35	0.82	2.35	2.33	2.24

TABLE 5: Number of writes to the connectors (w).

$e_{(1,2)}$	$e_{(2,3)}$	$e_{(2,12)}$	$e_{(2,13)}$	$e_{(1,3)}$	$e_{(3,4)}$	$e_{(4,5)}$	$e_{(5,6)}$	$e_{(6,7)}$	$e_{(7,8)}$	$e_{(8,9)}$	$e_{(9,10)}$	$e_{(10,11)}$	$e_{(11,12)}$	$e_{(12,13)}$
3	3	3	3	43255	44928	44928	44928	44928	15682	44928	44928	44928	44928	43255

be lower than 10%. Obviously the queue size can be increased subject to the availability of memory in the tiles.

Looking deeper into the distribution of the total execution time among the application tasks, middleware tasks, and the context switching, we obtained the results shown in Figure 9. In the case without the middleware, the critical node that determines the total execution time is n_3 for $bound \leq 2$ and n_1 for $bound > 2$. Since n_1 is mapped only one task (i.e., *src*), we do not see any time spent for context switching. As the connector bound is increased, n_1 becomes the critical node rather than n_3 due to the decreasing context switching penalty on n_3 .

In the case with the middleware, the critical node is n_2 for $bound \leq 10$ and n_5 for $bound > 10$. The change of critical node is again due to the decreasing context switching times when the size of the KPN queues is increasing.

7. Fault Tolerance Support

Having isolated the application tasks from the network interface, we believe it will be easier to implement fault tolerance mechanisms, mainly based on tasks migration. Below we describe the task migration support over the proposed middleware and list a number of self-checking patterns that are enabled by the task migration support. We give implementation details for the adoption of these patterns at KPN level. These patterns have been implemented and validated with SACRE framework.

The proposed patterns can be applied as run-time adaptations [4] enabling an adaptive degree of fault tolerance. They enable adaptive dependability levels for different parts of the application pipeline at the granularity of an application component. Besides conventional adaptation parameters such as voltage level and frequency, this allows to increase fault tolerance characteristic of the application in presence of excess power or resources. Typically, in self-adaptive systems, the application programmer provides a set of goals (e.g., performance, power) to be met by the application [5]. These goals are translated into parameters to be monitored by the self-adaptive platform. The adaptations are driven by a self-adaptation control mechanism that tries to meet the goals by monitoring those parameters. The proposed self-checking patterns, if applied at run time, enhance the self-adaptive platform to accept fault tolerance as a goal.

In our discussion, we mainly assume a functional fault model which considers a processing element as *working*, *nonworking*, or *partially working*. In case of a *working* processing element, the module is fully functional; in *partially working* ones, some errors have been detected, and the module is assumed to be faulty. However, the core has a relevant complexity and modularity (e.g., a processor with the floating point pipeline as well as with the fixed point one) and identification and confinement of a specific faulty subunit and degradation of the core to the surviving functions can be applied (see, e.g., [31]). However, the module can still be used, even if partial degradation of the functionality and of the performances must be taken into

account; for *nonworking* processing elements, the module is considered too faulty to be operative.

The fault tolerance patterns discussed in the following part of the section can be used to detect the operativeness of the processing element, as well as providing methods for detecting and correcting errors due to faulty behaviors of the components. Once the level of operativeness is detected, and depending on the faults detected, tasks migration can be used to move information from a nonworking (or a partially working) processing core to a working one.

7.1. Task Migration. In the case when a tile fails, the tasks mapped on that tile should be moved to other tiles. Therefore we will have a controller implementing a task remapping strategy. For now, we do not focus on this but rather highlight how the proposed middleware eases the implementation of task migration mechanisms.

Moving an application component from one tile to another requires the ability to start the task on the new tile, update the component mapping tables on each tile, create/destroy MW tasks for outgoing and incoming connectors of the migrated components, and transfer the tokens already available in the connectors of the migrated components along with those components. In case of a fault, the tokens in the queues pending to be forwarded by the middleware tasks in the failed tile may be lost along with the state of the task if it had any. Similarly, there may be some number of received flits that have not been reconstructed to make up a packet yet. We may need to put in measures to checkpoint the state of the task and the middleware queues. As a rollback mechanism, we should be able to transfer both the state of the tasks and the queues on the faulty tile to the new tiles. The flits already in the NoC buffers destined to the faulty tile should be rerouted to the new tiles accordingly. This may be easier to achieve if we implement the task-awareness feature in the NoC routers. Otherwise, it should be the NI or the router of the faulty tile that should resend those flits back in the network with correct destinations. We need to further analyze the scenarios according to the extent of faults (e.g., only the processing element is faulty or whole tile is faulty). However, thanks to the middleware layer, application tasks will not need to know that there has been a task migration.

7.2. Triple Modular Redundancy Pattern at Application Level. The run-time environment can adapt the dependability at the application level. This enables adaptive dependability levels for different parts of the application pipeline at the granularity of an application component.

In the case of a single component, parallel instances of the component are created on different cores along with multiplier components and majority voter components for each input and output ports, respectively, as shown in Figure 11. Multiplier component creates a copy of the incoming message for each redundant instance and forwards each copy to the input ports of those instances. Majority voter component reads a token from all of its input ports and finds out the most recurrent token and sends it to its output connector. If there is a different input token then this

FIGURE 11: Triple modular redundancy adaptation pattern applied at KPN level. KPN task C in (a) is replicated by three as shown in (b).

FIGURE 12: RFCS adaptation pattern applied at KPN level.

component can signal that the core producing that token is faulty. A timeout mechanism can also be put in place to tolerate when a core is faulty and no message is being received by a component.

7.3. Roll-Forward Checkpointing Scheme (RFCS) Pattern at Application Level. A lower power version of the TMR is the duplex with spare technique also referred to as Roll-Forward Check-pointing Scheme (RFCS) [32]. We propose to employ it for a KPN task as shown in Figure 12. In this scheme, the RFCS multiplier component forwards the copy of the token to the first two components (C_1 and C_2). After C_1 and C_2 execute, RFCS checker component reads the tokens from its first two input ports. If the tokens are the same, it forwards either one of the tokens and sends a *no-fault* token back to the multiplier. The multiplier keeps forwarding the tokens to its first two output ports as long as it reads a *no-fault* token before forwarding. If there is a fault in either of C_1 and C_2 , then the checker will forward one of the resulting tokens and signal back to the multiplier that there has been a fault by sending a *fault-present* token. If the multiplier receives such a token, it does not consume any tokens from its input port, instead, it forwards two copies of the last read token to C_2 and C_3 . After having sent the *fault-present* token, in the next round, the checker component reads the tokens from its second and third input ports. Comparing the results from the previous mismatch (between C_1 and C_2) and the current round (C_2 and C_3), the checker component can detect the faulty core.

7.4. Parallelization Pattern at Application Level. An example of a structural adaptation in order to meet performance and low-power goals is the parallelization pattern explained below.

Parallelization of a component is one type of structural adaptation that can be used to increase the throughput of the system as shown in Figure 13. This is done by creating parallel instances of a component and introducing a router before and a merger after the component instances for each of the input and output ports. A router is a built-in component in our framework that works in a round-robin fashion; this component routes the incoming messages to the parallel instances sequentially. The merger components simply merge the output messages from the output ports of the instances into one connector by reading again in a round-robin fashion from their input ports. For the general class of KPN applications, semantics require that the processes comply with the monotonicity property. The round-robin policy of the router and the merger preserves the ordering relation among tokens. However the condition for applicability of such an adaptation is the absence of intermessage dependencies.

7.5. Semiconcurrent Error Detection Pattern at Application Level. We propose to employ semiconcurrent error detection (SCED) [33] as a dependability pattern for self-adaptivity. This pattern is used on top of the parallelization pattern. It can be used in the case where there are already multiple parallel instances for performance reasons and we want fault tolerance on top of it. The basic idea is to have a redundant component instance (C_r) that is used to process the same token as one of the other parallel instances in a round robin fashion. We make a statistical assumption that fault rate and distribution of nominal input data values are such that with very high probability occurrence of a fault will be detected by the input data before occurrence of a second, independent fault. (In other words, coverage of all single faults in a unit is granted by the flow of nominal data within the time

FIGURE 13: Adaptation pattern for parallelization. Component in (a) is parallelized by three as shown in (b).

FIGURE 14: Semiconcurrent error detection at KPN level.

between two possible fault occurrences) Figure 14 shows the components and their interconnections. The *SCED router* component forwards a token from its input port to a different parallel instance just like the *router* component in the parallelization pattern. It also keeps track of an index variable that shows which is the instance currently being checked. Given N parallel instances, every N tokens, one of the instances is checked by sending the same token to the redundant instance. *SCED merger* component reads its input ports in order just like the *merger* component in the parallelization pattern. It also compares the results of C_r and the currently checked instance. If the results are identical, then both components are fault-free. If they are different, then it may be that either of them is faulty. If the results are again different in the next checking round, then the redundant instance is declared faulty. Otherwise it is the parallel instance that has been faulty in the previous checking round. The faulty result can be propagated to the rest of the system depending on the application characteristics. If no fault propagation is allowed, the results will be buffered internally in the *SCED merger* component until the faulty component is identified correctly. Then the results can be forwarded to the output of the *SCED merger* by choosing the results of the nonfaulty component.

8. Conclusion and Future Work

In this work, we proposed an active middleware layer to accommodate KPN applications on NoC-based platforms. Besides satisfying KPN semantics, the middleware allows application components to be platform independent with regard to the on-chip communication infrastructure. The

middleware is solely based on `MPI_send()` and `MPI_recv()` communication primitives, thus it does not require any modification to the NoC platform. The middleware is an initial step towards implementing a self-adaptive run-time environment on the NoC platform. In order to realize a functional simulation of the system proposed, we integrated our SACRE tool with a NoC cycle-accurate simulator, and we evaluated the overhead, in terms of computational time and total data traffic, associated to the use of the middleware in the case of a JPEG case study. The results show that the overhead in both metrics can be lower than 10% depending on the chosen bound of the connectors and the size of tokens being transferred at the application level.

Moreover, we proposed application level fault tolerance patterns, namely, TMR, RFCS, and SCED patterns, that can be applied on a KPN application regardless of tasks being run on programmable or nonprogrammable cores as long as the middleware and the adaptation mechanisms allow us to treat them in the same way while modifying the application graph. The patterns can be applied to a given KPN application graph automatically by an adaptation controller and in cooperation with the self-adaptive run-time environment. Thus they set the application programmer free from having to care about dependability concerns.

Acknowledgments

This work was funded by the European Commission under Project MADNESS (no. FP7-ICT-2009-4-248424). 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. The authors would like

to thank Mariagiovanna Sami for her valuable comments on this work.

References

- [1] G. De Micheli and L. Benini, *Networks on Chips: Technology and Tools*, Morgan Kaufmann, Burlington, Mass, USA, 2006.
- [2] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceedings of the IFIP Congress (Information Processing '74)*, J. L. Rosenfeld, Ed., pp. 471–475, North-Holland Publishing Company, New York, NY, USA, 1974.
- [3] O. Derin and A. Ferrante, "Simulation of a self-adaptive runtime environment with hardware and software components," in *Proceedings of the ESEC/FSE Workshop on Software Integration and Evolution at Runtime (SINTER '09)*, pp. 37–40, ACM, New York, NY, USA, August 2009.
- [4] O. Derin and A. Ferrante, "Enabling self-adaptivity in componentbased streaming applications," *SIGBED Review*, vol. 6, no. 3, special issue on the 2nd International Workshop on Adaptive and Reconfigurable Embedded Systems (APRES '09), 2009.
- [5] O. Derin, A. Ferrante, and A. V. Taddeo, "Coordinated management of hardware and software self-adaptivity," *Journal of Systems Architecture*, vol. 55, no. 3, pp. 170–179, 2009.
- [6] "Noxim NoC simulator," <http://noxim.sourceforge.net>.
- [7] E. Carara, A. Mello, and F. Moraes, "Communication models in networks-on-chip," in *Proceedings of the 18th IEEE/IFIP International Workshop on Rapid System Prototyping (RSP '07)*, pp. 57–60, IEEE Computer Society, Washington, DC, USA, 2007.
- [8] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere, "System design using Kahn process networks: the compaan/laura approach," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '04)*, vol. 1, p. 10340, Paris, France, February 2004.
- [9] 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, pp. 233–270, 2002.
- [10] 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. 1–39, 2008.
- [11] 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.
- [12] 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 *Proceedings of the IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia '09)*, pp. 35–44, IEEE, Grenoble, France, 2009.
- [13] "Multicore associations communication api," <http://www.multicore-association.org>.
- [14] "A high performance message passing library," <http://www.open-mpi.org/>.
- [15] G. M. Almeida, G. Sassatelli, P. Benoit et al., "An adaptive message passing MPSoC framework," *International Journal of Reconfigurable Computing*, vol. 2009, p. 20, 2009.
- [16] 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 '2009)*, November 2009.
- [17] E. Beigne, F. Clermidy, P. Vivet, A. Clouard, and M. Renaudin, "An asynchronous noc architecture providing low latency service and its multi-level design framework," in *Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems*, pp. 54–63, IEEE Computer Society, Washington, DC, USA, 2005.
- [18] I. Koren and C. M. Krishna, *Fault Tolerant Systems*, Morgan Kaufmann, San Francisco, Calif, USA, 2007.
- [19] A. Avizienis, "The n-version approach to fault-tolerant software," *IEEE Transactions on Software Engineering*, vol. 11, no. 12, pp. 1491–1501, 1985.
- [20] M. R. Lyu, *Handbook of Software Reliability Engineering*, McGraw-Hill, Hightstown, NJ, USA, 1996.
- [21] C. Fuhrman, S. Chutani, and H. Nussbaumer, "A fault-tolerant implementation using multiple-task triple modular redundancy," in *Proceedings of the IEEE International Workshop on Factory Communication Systems (WFCS '95)*, pp. 75–80, October 1995.
- [22] M. Rebaudengo, M. S. Reorda, and M. Violante, "A new approach to software-implemented fault tolerance," *Journal of Electronic Testing*, vol. 20, no. 4, pp. 433–437, 2004.
- [23] E. L. Rhod, C. A. Lisboa, L. Carro, M. Sonza Reorda, and M. Violante, "Hardware and software transparency in the protection of programs against SEUs and SETs," *Journal of Electronic Testing*, vol. 24, no. 1–3, pp. 45–56, 2008.
- [24] J. Ceponis, E. Kazanavicius, and A. Mikuckas, "Fault tolerant process networks," *Information Technology and Control*, vol. 35, no. 35, pp. 124–130, 2006.
- [25] J. Čeponis, E. Kazanavičius, and L. Čeponiene, "Handling multiple failures in process networks," *Information Technology and Control*, vol. 37, no. 1, pp. 19–25, 2008.
- [26] T. M. Parks, *Bounded scheduling of process networks*, Ph.D. thesis, University of California, Berkeley, Calif, USA, December 1995.
- [27] S. Murali and G. De Micheli, "Bandwidth-constrained mapping of cores onto NoC architectures," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '04)*, vol. 2, pp. 896–901, February 2004.
- [28] L. Thiele, I. Bacivarov, W. Haid, and K. Huang, "Mapping applications to tiled multiprocessor embedded systems," in *Proceedings of the 7th International Conference on Application of Concurrency to System Design (ACSD '07)*, pp. 29–40, July 2007.
- [29] T. T. Ye, L. Benini, and G. De Micheli, "Packetized on-chip interconnect communication analysis for mpsoC," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '03)*, vol. 1, p. 10344, IEEE Computer Society, Washington, DC, USA, 2003.
- [30] "Sesc simulator," <http://sesc.sourceforge.net>.
- [31] K. Reick, P. N. Sanda, S. Swaney et al., "Fault-tolerant design of the IBM Power6 microprocessor," *IEEE Micro*, vol. 28, no. 2, pp. 30–38, 2008.
- [32] D. K. Pradhan and N. H. Vaidya, "Roll-forward checkpointing scheme: a novel fault-tolerant architecture," *IEEE Transactions on Computers*, vol. 43, no. 10, pp. 1163–1174, 1994.
- [33] A. Antola, F. Ferrandi, V. Piuri, and M. Sami, "Semiconcurrent error detection in data paths," *IEEE Transactions on Computers*, vol. 50, no. 5, pp. 449–465, 2001.