Journal of Systems Architecture 59 (2013) 1032–1046

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

A unified execution model for multiple computation models of streaming applications on a composable MPSoC

Ashkan Beyranvand Nejad [a,*], Anca Molnos [b], Kees Goossens [c]

[a] Computer Engineering Lab., Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands
[b] CEA-Leti, Grenoble, France
[c] Electronic Systems Group, Eindhoven University of Technology, Den Dolech 2, 5612 AZ Eindhoven, The Netherlands

ARTICLE INFO

Article history:
Available online 26 July 2013

Keywords:
Execution model
Data-driven models of computation
Real-time applications
Composable system
Multi-processor system-on-chip

ABSTRACT

In this paper we propose a unified model of execution that aims to fill the abstraction level gap between the primitives of models of computation and the ones of an MPSoC. This model targets a composable MPSoC platform and supports the sequential, Kahn process networks, and dataflow models. Our model comprises of (1) execution operations implementing the primitives in the models of computation, and (2) a time model of execution of streaming applications on a composable platform. We implement these models of computation with the model of execution, and discuss the trade-offs involved. Case studies on an FPGA prototype of the composable MPSoC demonstrate how the model of execution actually works on a real platform. Furthermore they indicate that multiple applications modeled in KPN and dataflow run composably on the platform.

1. Introduction

In recent years the trend in embedded systems is to execute an increasing number of applications simultaneously. Applications are functionally-independent software units that are developed by possibly different parties. Multi-processor systems on chip (MPSoCs) [1] are platforms that can offer concurrent execution of multiple applications. MPSoCs embed a large number of resources, e.g., processing elements, memories, and user interface peripherals, on a single chip.

In embedded systems, many applications are streaming, i.e., data-driven, such as audio and video codecs, or networking applications. They may have soft, firm, or non real-time requirements. A firm real-time (FRT) application must never miss a deadline, whereas a soft real-time (SRT) one may occasionally miss a deadline, and non real-time (NRT) ones are completely timing relaxed. Consequently, these three application domains require different design strategies and models, that could truly express their functionality, implement and verify their timing requirements.

Currently, most applications are still first designed and tested using an imperative, sequential, model of computation (MoC), typically C. Multiple applications may execute in parallel on an MPSoC platform. However, to fully exploit the computation power of an MPSoC, the parallelism is not restricted to the application level, but each application is further split in a number of concurrent tasks [2]. In the literature, various parallel models of computation are proposed for streaming applications [3]. Each of these models has its own properties that makes it suitable for various application domains [4]. The two important properties of these models are (i) expressiveness, i.e., the level of computation primitives that a model of computation offers to express the applications' functionality, for example, how algorithms that are dependent on the values of input data could be expressed by the model of computation; and (ii) analyzability, i.e., how amenable the model is for accurate timing analysis, for example, in a real-time model, once started, tasks may execute without any blocking, and therefore, a worst-case bound for their execution time could be estimated. Usually, there is a trade-off between the analyzability and the expressiveness of a model of computation.

The comparison between the relative analyzability and the relative expressiveness of the most widely used models of computation (MoCs) for data-driven applications is presented in [5]. Here, Fig. 1 visualizes the result of this comparison. In general, sequential models of computation have no restrictions in using the primitives of programming languages such as C, and therefore they can be highly expressive in modeling different behavior of applications, except for parallelism, while the analyzability of the models may be lost. If the sequential model of computation is restricted to a subset denoted as *Nested Loop Programming* (NLP), an initial step towards an intra-application parallel implementation is made.

An NLP application can be manually or automatically translated into parallel models of computation [6], such as *Kahn Process*

* Corresponding author. Tel.: +31 (0) 15 27 83644; fax: +31 (0) 15 2784898.
E-mail addresses: A.BeyranvandNejad@tudelft.nl (A. Beyranvand Nejad), Anca.Molnos@cea.fr (A. Molnos), K.G.W.Goossens@tue.nl (K. Goossens).

1383-7621/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.sysarc.2013.07.006

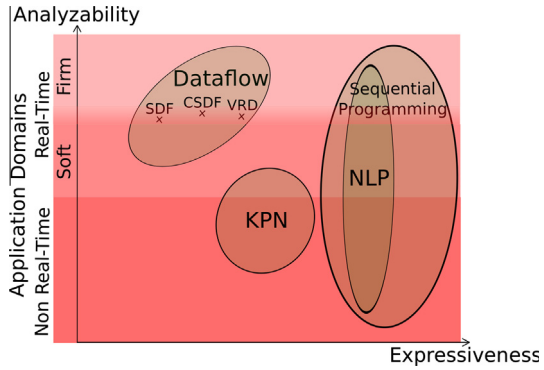


Fig. 1. Relative analyzability vs. expressiveness of the common models of computation for streaming applications, and the application domains for which each of the models is suitable.

Network (KPN) [7], and *dataflow* [8], and can be used in FRT, SRT, and NRT application domains. Furthermore, KPN is more expressive than many variants of the dataflow, e.g., Static Dataflow (SDF), Cyclo-Static-Data-Flow (CSDF), and correspondingly, less analyzable, which limits KPN to be only applicable for SRT and NRT application domains. Dataflow has different variants with different level of analyzability and expressiveness. For example CSDF and SDF suit the FRT and SRT application domains [9]. Thus, by supporting NLP, KPN, and dataflow models in a platform, a wide range of analyzability and expressiveness for data-driven applications is covered.

To reduce implementation cost, applications may share MPSoCs resources. Such resource sharing cause inter-application interference. This results in two problems, (i) the temporal behavior of the applications is influenced by the interference, and subsequently, deadline of RT applications may be missed, or even worse, an application monopolizes a shared resource, other applications may get completely blocked, and (ii) it is not possible to verify an application's functional and temporal behavior without considering all (or at least the worst case) interference scenarios with other applications. Depending on the number of applications, the number of these scenarios may be large.

Composability is proposed and advocated to provide independent execution and to alleviate system-wide, monolithic verification [10,11] by avoiding inter-application interference. An MPSoC is composable if an application's timing and functionality is not influenced by the behavior of other applications. Consequently, applications can be designed, verified and executed in isolation, without invalidating their designed properties. In these systems, all the shared resources are designed in such a way that they may be virtualized to achieve applications isolation. For this purpose, on a shared processor, a composable operating system provides a virtualization layer between the applications' interface and the bare hardware.

1.1. Problem statement

The applications executing concurrently on one MPSoC may be implemented in different models of computation, with each model having its own primitives with different level of expressiveness, as discussed above. On the other hand, the MPSoC platform provides low-level execution primitives, e.g., memory load and store, or inter-processor data transfer. To implement an application, the computation and communication primitives of a model of computation should be mapped to the execution primitives of the platform. However, the primitives of the models of computation and the execution platform are at different levels of abstraction which causes a gap in the implementation process of applications. Therefore, intermediate execution operations are needed to map the primitives of different models of computation to the basic primitives of the platform.

Furthermore, real-time applications require temporal verification in order to guarantee that their deadlines are always met. The performance of a real-time application is relative to the real, physical world, or, in other words, the deadlines of an application are relative to physical time. However applications share resources, which means that an application is suspended when another one uses the resource, e.g., when it is swapped out from a shared processor. As a result, the application executes in a virtual time that is experienced as continuous. However the physical time when it actually executes is no longer continuous. Therefore, to verify the timing properties of an application the following are required. First, the application timing behavior, in its virtual, continuous time, should be bounded. This should be ensured by the implementation of the application. Second, a time model that defines the effect of resource sharing between multiple applications, i.e., maps the virtual time of an application to the physical time, is required. This time model should be defined by the model of execution.

To summarize, the problems that we address in this paper are: (i) the need for an intermediate level of abstraction to fill the gap between the execution operations of the platform and the primitives of different models of computation, and (ii) the need for a time model of the execution platform that enables the designers to develop and verify the applications models, with respect to their physical timing properties, when multiple applications share resources on the same MPSoC platform. Furthermore, the resulted system should be composable, to enable independent application development and verification.

1.2. Contributions

When applications are implemented in different models of computation, there are two approaches to fill the previously mentioned abstraction gap: (i) to define one set of execution operations for every model of computation, and (ii) to identify a common set of execution operations (e.g., basic communication primitives) onto which the operations of all models of computation (e.g., read/write, consume/produce) can be implemented. The second approach is more generic and efficient in the sense that it aims to minimize the variety of execution operations that should be realized by the platform. Following this approach, in this paper, common operation primitives of different models of computation are defined and implemented in a unified *model of execution* (MoE).

The abstraction levels of an embedded system are presented in Fig. 2. At the highest layer, an application belongs to one of the

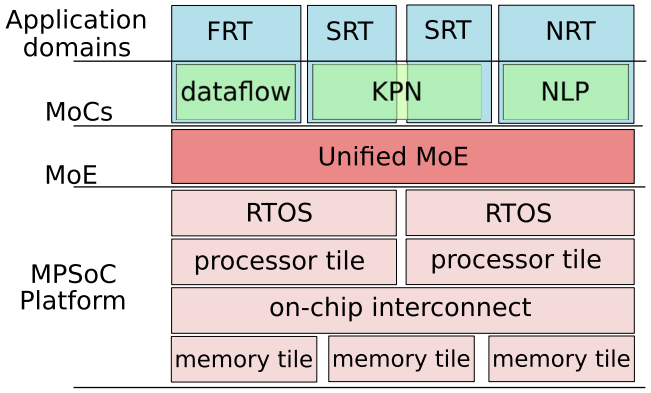


Fig. 2. Abstraction levels of an embedded MPSoC.

domains of FRT, SRT, or NRT. At the second layer, the application is modeled in one of the models of computation that suits the timing constraints and is expressive enough to model the behavior of the application. The application might be split into a number of concurrent tasks manually or automatically by using the existing tools such as PNGEN [12] and Compaan [13]. The application's model is then integrated with models of all the MPSoC resources that are used by the application and a meta model of the system is made [14]. Note that the MPSoC resource models include the resource sharing effects and potential overheads of intermediate software layers of the platform. This model is analyzed manually or automatically using the existing tools such as SFD3 [15] to partition and map the application on different processor cores, communication channels, and memory blocks, such that the timing requirements of the application are guaranteed. In Fig. 2, the KPN application is an example of the applications that are mapped on multiple processors.

The model of execution is the intermediate layer that fills the abstraction gap between the application's model of computation and the implementation on an MPSoC platform. The model of execution defines a set of computation, communication and scheduling operations that are required to execute an application expressed in a model of computation. For this purpose, the model of execution implements the primitives of the models of computation and maps those primitives on an execution time model of the platform.

On every processor tile, the model of execution is realized as a software wrapper which includes a high-level application programming interface (API) that abstract the low-level system calls of the underlying real-time operating system (RTOS) and a set of programming templates. For instance, the data transfer between the memory locations on different tiles are handled by the RTOS using low-level direct memory access (DMA) send and receive operations. The model of execution hides these low-level operations from the applications' perspective and provides high-level API for the communication and the synchronization between the tasks of the applications that are mapped on two different processor tiles. Moreover, formatting the applications in the programming templates simplifies the implementation of the applications given in the different models of computation.

The hardware infrastructure of the platform consists of a number of processor tiles (each including one core and local memory) and a number of memory tiles communicating via an on-chip interconnect. Following the asymmetric multiprocessing (AMP) method [16], an instance of an RTOS runs on each processor tile. Every instance of the RTOS schedules the applications mapped on the corresponding processor tile independently. Therefore, here we focus on execution of applications on one processor tile and how the inter-/intra-processor communication and synchronization between the application's tasks are implemented with the model of execution.

In this context, the contributions of this paper are fourfold. First, we propose a unified execution model that implements all of the NLP, KPN, and dataflow models of computation. The model of execution (MoE) is formalized in three steps, (i) the execution operations implementing the models of computation' primitives are introduced, (ii) a time model of applications executing on a composable platform is defined, and (iii) different options of mapping the execution operations to the time model are defined. Following that, the second contribution is to implement the models of computation with the model of execution, and to discuss the trade-offs between different implementation options. Third, we propose an API that implement the unified model of execution on top an existing composable platform in [11,17]. Finally, we experimentally demonstrate that a set of applications modeled in NLP, KPN, and dataflow run simultaneously, composably on the MPSoC platform.

1.3. Outline

The rest of this paper is organized as follows. Related work is discussed in Section 2. Section 3 gives an overview of the application computation models. The model of execution is proposed and formalized in Section 4 for NLP, KPN, and dataflow. Section 5 discusses the trade-offs involved in mapping the models of computation to the variants of the model of execution. Section 6 presents the implementation of the model of execution on a composable MPSoC platform, and the experimental results are presented in Section 7. Finally, Section 8 concludes this paper.

2. Related work

Existing work falls into four categories: (i) composable systems on chip, (ii) mapping and implementation of the various models of computation, specifically the KPN and dataflow, (iii) design strategies for multiple applications execution on MPSoC platforms, (iv) high level models of computation refinement towards different models of execution. In what follows, we position our approach with respect to the existing work in these categories.

Generally, the existing execution models are either tailored to a single model of computation, or assume no parallel execution of multiple applications on a processor. Moreover, targeting a composable system distinguishes our approach from the similar existing work. Various definitions of composability exist [18,9,19]. Our definition however is more restrictive in that inter-application interference is completely prohibited at cycle-level. The advantage is that a mix of FRT, SRT and NRT applications can be easily, independently designed, verified, and integrated on the same MPSoC platform.

Several execution platforms for KPN applications were proposed [9,20–26]. Except the work presented in [9], none of them targets MPSoC platforms. The authors of [9] propose an MPSoC platform that supports multiple real-time applications. The system performance is estimated using the applications individual timing profiles and a model of the inter-application interference. When the applications are developed by different parties and not all of them are available at design time, it is not possible to come up with such estimation. Thus, this approach is quite restrictive comparing to our technique that targets a composable system which enables design, verification and integration of the applications in isolation. Furthermore, the approaches in [25,26] differ from ours in the sense that in [25] KPN processes are scheduled and executed on hardware reconfigurable accelerators, and in [26] not multiple applications may execute concurrently on the platform.

In case of dataflow models, applications performance can be accurately analyzed using several dataflow models [27–29]. Thus dataflow is used to express real-time applications executed on MPSoCs [19,30]. All these approaches allow the design of real-time applications, however the analysis requires bounds on the execution time of each task or preemption in bounded time. This is not generally the case for non-real-time applications, thus their integration on a common platform is not straightforward. The authors of [19] use an MPSoC similar to our platform, and target a system that permits reasoning about the worst case overall behavior of applications when they are analyzed in isolation. This means that the running applications can affect each other's timing behavior and the worst case still holds true. This definition of composability is very similar to the ones in [9,30], and, as mentioned above our definition is more restrictive than theirs.

Typically, the programming and implementation models of embedded applications start from a highly abstract model and refine the model to less abstract implementation models. For-

SyDe [31] provides a disciplined mixture of models of computation for embedded systems designs. ForSyDe starts from a high level functional description of applications using Haskell as the modeling language, and refines it step by step to an implementation model. Thus the approach does not necessarily target an MPSoC platform. In this approach, the refined implementation model may be for example another model of computation such as a KPN, or a dataflow. We can consider its generated model as an equivalent for our model of execution, however, here we are explicit about the implementation of the models of computation with the model of execution and we target only the KPN and dataflow models. Our model of execution not only represents the execution operations of these models but also maps the operations to the time model of the composable platform.

Furthermore, PTIDES [32] and Giotto [33] are models of computation that target real-time applications. PTIDES focuses on the automotive application domains that include sensors and actuators and proposes the execution strategy for timed models [34]. It only supports applications with hard real time requirements, and introduces a local notion of the real, physical time for the applications. Their definition of the local time is similar to the logical execution time of applications in [35]. Our model of execution also proposes a logical (virtual) time for the applications, while each application time is totally isolated from the others. Giotto is a time-triggered language for embedded programming that targets embedded control applications. It achieves time predictability but no composability, and the application timing may influence each others' timing properties. Giotto does not specify where, how and when the task are scheduled and executed, an the model is generic to be implemented by different models of execution. Giotto and PTIDES aim to provide independent programming models (models of computation) from the underlying platform, and their models are not data-driven.

Moreover, CASSE [36] proposes a high level execution model for simulating the applications that are functionally modeled in KPN. It bridges the gap to system implementation by refining the KPN model, and enabling transaction-level simulation at different abstraction levels in the model refinement procedure. In the context of hardware/software co-design, the work presented in [37] deals with parallel programming models to abstract both hardware and software interfaces in the case of heterogeneous MPSoC design. The authors of this work discuss different models of application programming interface (API) and how an MPSoC simulation may benefit from high level models. Unlike these approaches that aim to simplify the functional simulation of applications, our model of execution bridges the gap between the actual MPSoC platform and the model of computation. Using our time model, the designers can verify the requirements of each application in isolation at design time, and using the operations of the model of execution, different data-driven models are implemented easily on the actual MPSoC platform.

3. Models of computation

Traditionally, applications are initially designed using a sequential model of computation (MoC) in a high level programming language, such as C. The Nested Loop Programming (NLP) is a subset of a sequential MoC. NLP models an application as a number of loops over single assignment basic *functions*, as presented in Fig. 3a. NLP can be automatically transformed into parallel models of computation [6,13], e.g., KPN and dataflow. The KPN and dataflow models are networks of autonomous and concurrent *nodes*, referred to as *processes* in KPN and as *actors* in dataflow terminology [38]. A node corresponds to one or more function calls in the NLP model of the application, and it is a functional mapping from input streams to output streams (corresponding to the function's arguments). Each node executes for a *possibly infinite* number of activations. Nodes communicate along unidirectional channels by means of data *tokens* that are sent and received in a First-In-First-Out (FIFO) order, as presented in Fig. 3b. In other words, in the KPN and dataflow models of computation, the synchronization between a producer node and a consumer one is done implicitly via the FIFO operations.

A KPN process body, illustrated in Fig. 3c, consists of a sequence of `read`, `compute`, and `write` operations. These operations may be interleaved in any order, and a process may read or write an arbitrary number of tokens from or into a FIFO. Although theoretically the FIFO sizes are infinite, practically, each FIFO is implemented with a bounded capacity [22]. A process blocks on a read or write when the FIFOs does not have enough input data or output space, respectively. A KPN process is activated once and it potentially executes for a(n) (in)finite number of iterations, and the process itself should implement its iterative execution, as illustrated in Fig. 3c with the main loop inside the body.

A dataflow actor body is a sequence of `consume`, `compute`, and `produce` operations, in this strict order, as presented in Fig. 3d. A firing rule specifies, for one actor activation, for each incoming and outgoing edge, the number of input tokens consumed and the number of tokens produced, respectively. Once the firing rule is satisfied, an actor executes its entire body without blocking. The actors execute for an infinite number of activations as long as their firing rules are satisfied. In dataflow, each activation of an actor corresponds to one iteration.

Different variants of dataflow models exist, e.g., Static Dataflow (SDF), Cyclo-Static Dataflow (CSDF) [27], Variable Rate Dataflow (VRD) [28], some of which are analyzable, i.e., a worst case timing of each of its actor can be calculated. This is due to the fact that after a dataflow actor firing rules are satisfied, the input an output tokens are ready for the entire execution of it computation without any blocking, and therefore, a bound on computation time of the actor can be easily calculated. Assuming that the MPSoC platform is predictable, the communication time through the interconnect to access the distributed memory locations in the system can be also bounded. Therefore, existing formalisms can analyze an application and derive an end-to-end latency, throughput, and buffer sizes for it [29]. Our target platform can execute all three mentioned variants of the dataflow, however, in this paper, we focus on one of the analyzable ones, i.e., CSDF.

KPN and dataflow have different properties that make them suitable for different application domains. Some variants of the dataflow are suitable for the FRT domain that demands timing analysis, since the actors execute their entire body without blocking and the communication through FIFOs is predictable (i.e., it is known at design time how many tokens are produced and consumed by the actors). However, dataflow is not expressive enough to model dynamic application behavior, e.g., the production and consumption of a data-dependent number of tokens on a channel. Such behavior is common in the signal processing domain, e.g., variable-length encoding and decoding. KPN is a suitable model for such dynamic applications, as it allows arbitrary production and consumption rates and arbitrary interleaving of communication and computation inside a process. On the other hand, KPN is not amenable to exact timing analysis required for FRT applications, because KPN allows arbitrary interleaving of computation and communication in a process body. As a result the time that is spent waiting for data in communication can block the execution of the process body, or data-dependent computation causes unpredictable execution time of the process. However, this unpredictable timing behavior may be analyzed statistically to come up with a probabilistic timing behavior that may cause occasional deadline misses at run-time [9], which is acceptable for SRT appli-

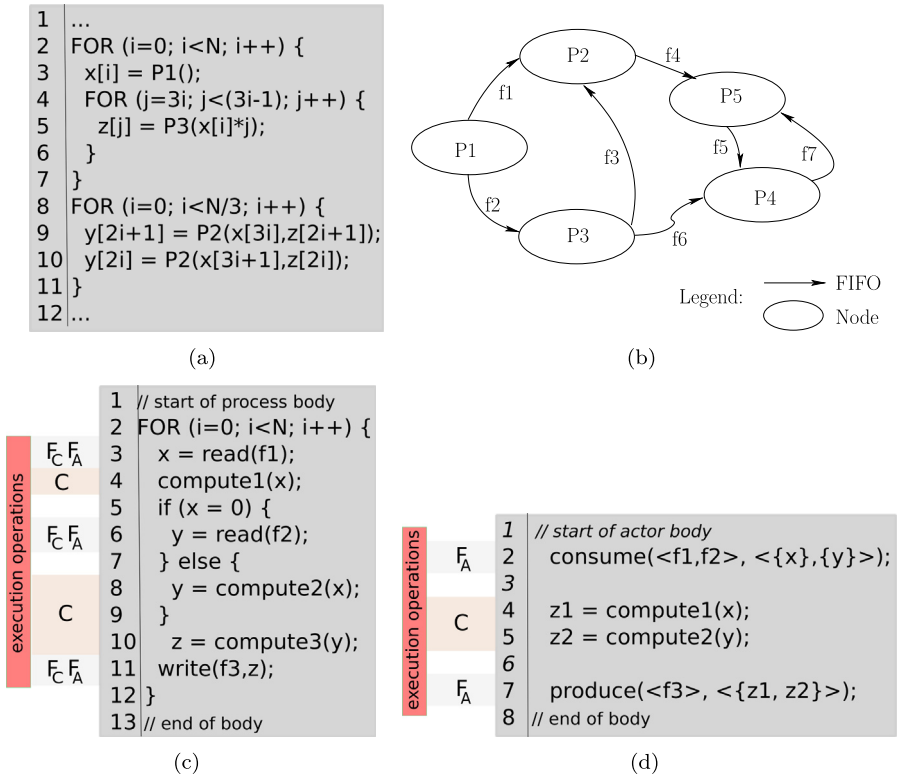


Fig. 3. Application computation models: (a) a Nested Loop Programmed (NLP) application, (b) an application task graph, (c) a KPN process example code, (d) a dataflow actor example code.

cations. Thus KPN can only fit NRT and SRT applications. Essentially, the execution of both KPN and dataflow models on an MPSoC platform enables our model of execution to support a wide range of application domains as illustrated in Fig. 1.

The execution operations corresponding to the models of computation are illustrated in Fig. 3c and d. These operations will be introduced and explained in the next section.

4. Model of execution

A *model of execution* (MoE) defines a set of operations that implement specific model(s) of computation, and a time model of applications executing on the MPSoC platform. An application is expressed in a model of computation, e.g., KPN, dataflow, which includes the operations for *computation* and *communication*, e.g., `read`, `produce`. Thus these operations should have equivalents in the execution model. Moreover, when multiple applications sharing resources, the model of execution should also define *scheduling* operations and the time at which these operations execute. In this section, we first define the execution operations of our model of execution. Following that, we introduce the time model of applications' execution on a composable MPSoC, and finally, we discuss the available design options in scheduling operations.

Table 1
List of the execution operations.

Operation	Category	Description
C	Computation	Performs the computational functions
F_C	Communication	Checks if a FIFO has available data or space
F_A	Communication	Accesses the data or space of a FIFO
S_A	Scheduling	Selects an application to be scheduled
S_T	Scheduling	Selects a task to be scheduled

4.1. Execution operations

The execution operations can be categorized as: (i) computation, (ii) communication, and (iii) scheduling operations. Our unified model of execution identifies and implements the common operation primitives required for execution of NLP, KPN and dataflow in these three categories, as presented in Table 1. The implementation of the models of computation by our model of execution, using the execution operations, are represented with a regular language sequence.[1] We first abstract from scheduling and present the set of execution operations and how our model of execution implements each of the NLP, KPN and dataflow models. Then, we detail the scheduling for each of these models.

A computation operation, C, is defined as the sequence of all instructions, except the instruction for communication, that implements the task's functionality, and it is usually between two consecutive communication operations. The communication between an application's tasks can be inter- or intra-processor-tile, depending on the tasks' mapping on the tiles. In an MPSoC platform with distributed shared memories, the difference between these two inter-tasks communication scenarios is in the actual memory locations that the FIFOs' data structures are implemented. Our model of execution however abstracts from these two scenarios and proposes general FIFO-based communication method which hides the details of FIFOs' implementation from the perspective of the applications. In this method, a FIFO requires administration and access operations. A FIFO administration operation is a space- or data-*check* denoted with F_C. A FIFO buffer *access*, i.e., placing/retrieving data into/from the buffer, is represented as F_A. F_A may be performed only after making sure that space or data exist, via F_C.

[1] If A and B are two operations, the regular expression language is defined as follows: $A\epsilon = \{A\}, (A+B) = \{A, B, AB\}$, $A^2 = \{AA\}$, $A^{[1,\infty)} = \{A, AA, AAA, \ldots\}$, and $A^{[0,N]} = \{\epsilon, A, AA, \ldots, A^N\}$.

The NLP model is mapped on the target execution platform as a simple application with one task and no FIFO. KPN processes and dataflow actors are each implemented as tasks, and the inter-process and inter-actor communication is implemented through FIFO. A KPN process activation or dataflow actor firing corresponds to a task iteration in the model of execution. The similarity between these two models stops here. We define a task's status as *eligible* if it can execute. If a task is not eligible, its status is *blocked*. A KPN process is initially eligible by default, i.e., the body of the process starts executing, until it gets either blocked on a FIFO read/write or it is finished. In other words, a KPN process (or its corresponding task) is activated once, and the (in)finite iteration over its operations can be implemented with a loop inside the process body. A dataflow actor is eligible if it has enough data and space in the input and/or output FIFOs, respectively. The task corresponding to an actor is activated when its firing rule is satisfied, and therefore, it can fire for infinite number of iterations.

For a dataflow actor, the corresponding task's status is unchanged for an entire iteration. A KPN process may immediately start (is eligible), and it blocks whenever it executes a `read` or `write` for which there is not enough data and/or space. Once it has started an iteration, a KPN task is not guaranteed to finish it without blocking, thus the task status is not fixed for an entire iteration.

KPN `read` and `write` operations require both FIFO check and FIFO access, whereas the dataflow `produce` and `consume` operations require only FIFO access. Formally, `read` and `write` operations are implemented as $F_C^{[1,\infty)}F_A$ (where F_C^{∞} models that the task may wait for data/space infinitely), and `consume` and `produce` operations as F_A. In the KPN model the `read` and `write` operations may be arbitrarily interleaved with `compute`, $\left(C + F_C^{[1,\infty)}F_A^{[0,1]}\right)^{[0,M]}$. Here, F_C^x represents checking a FIFO x times, and $\left(C + F_C^{[1,\infty)}F_A^{[0,1]}\right)^{[0,x]}$ models the finite iterations of a process for x times, where $x \in [1,\infty)$. In dataflow the execution order is strict, starts with all `consume` operation, i.e., $F_A^{[0,N]}$, continues with the `compute`, C, and ends with all `produce` operation, i.e., $F_A^{[0,N]}$. These correspondence of each MoC's operation with an execution operation is also illustrated in Figs. 3c and 3d beside the KPN and dataflow pseudo code lines.

Traditionally, the inter- and intra-application parallelization in the models of computation implies separate scheduler implementations for each level of the processor scheduling, i.e., inter-application and intra-application scheduling [39]. The inter-application scheduler selects only the application that executes next on the processor. The execution operation of inter-application scheduling is denoted with S_A. The intra-application (task) scheduler determines the next task of the selected application to execute. We define S_T as the execution operation that selects a task according to a given policy, e.g., time-division multiplexing (TDM) and Round-Robin. In KPN any policy can be used to schedule a task although scheduling an eligible tasks is more reasonable. Thus scheduling a KPN task can be implemented as an S_T operation. In dataflow the task scheduler has to first find an eligible task, thus it selects a task, S_T, and afterward, a check of each of its FIFOs, $F_C^{[0,N]}$, repeatedly, formally resulting in $(S_T F_C^{[0,N]})^{[1,N]}$. If an eligible task is not found, the *idle* task is scheduled.

4.2. Time model

To execute applications, the processors of the MPSoC perform specific operations triggered by clock ticks. The clock ticks is generated based on the *real time* which is the wall clock time that refers to the global, real, physical time. Moreover, when multiple applications sharing a processor, an application's view on real time, i.e., when it actually executes, is not continuous. The application is suspended when another application utilizes the processor. Thus, an application actual time consist of a number of discrete time slots. Fig. 4a presents the virtual time of three applications executing on a shared processor. An application's view on the set of slots at which it executes, is continuous and form the *virtual time* of the application. In Fig. 4a, every application has its own virtual timeline.

In the time slot that an application is allowed to utilize the processor, the virtual time of the other applications is frozen. Therefore, the virtual time of an application is completely disjoint from the virtual time of other applications, on the same processor. The separation between the virtual time of applications is not enough for a composable system. Composability implies that in a given platform the virtual time of every application should always map uniquely to the fixed moments of the real, physical time, independent from the other applications. When an application is mapped over several processors, it receives virtual time on each of the processors. The virtual time of the application on one processor also maps uniquely to the real time, independent from the virtual time of the application on the other processors.

To implement these, in practice the time on each processor is split in fixed-duration quanta, in a time-division multiplexing (TDM) fashion [40]. Time quanta are denoted as application slots; each slot is assigned to (at most) one application. Fig. 4b shows the TDM slot allocation for our example. The TDM period, which we refer to as the system period, is a repetitive real-time duration that is formed by a number of consecutive real-time slots. The time necessary to switch between two different consecutive application slots, is not included in the applications' virtual time, and is referred to as the *system time*. This time overhead is typically negligible comparing to the applications time slot, and therefore it is not shown in Fig. 4a. However, in a composable system in which an application's temporal behavior should be totally independent of the other applications, the overhead not only should be bounded but also should be always fixed [40]. In this way, the starting and finishing time of the applications' slots are always guaranteed to be at the same moment in the real time, and therefore, the execution time of an application is always fixed and independent from the others.

During its virtual time an application may initiate communication with other processor- or memory-tiles. Time models for interconnects and memory tiles exist in the literature [41,42], hence we do not discuss them further. However it is important to stress that the time models of different resources, e.g., processor and interconnect, are independent by design, because the considered MPSoC is predictable and composable [43]. In such an MPSoC, the time models of different resources are required to be independent, to ease the temporal analysis effort. To design a real-time application, all these models should be taken into account by analysis framework, as mentioned earlier.

4.3. Mapping the execution operations to the time model

Considering the time model proposed in Section 4.2, the execution operations may be performed by the processor in system or application time. In the system time, in which the applications may be switched, the application scheduling operation executes. However, after an application is scheduled, the task scheduler can be also called in the system time to select the task to execute in the application's coming time. Formally, the operations that can execute in this time, consist of S_A, and possibly, S_T, and F_C.

Moreover, an application's operations should execute in its own time (application time), where the operations consist of computation, C, and FIFO access, F_A, and possibly the task selection, S_T, and FIFO check, F_C.

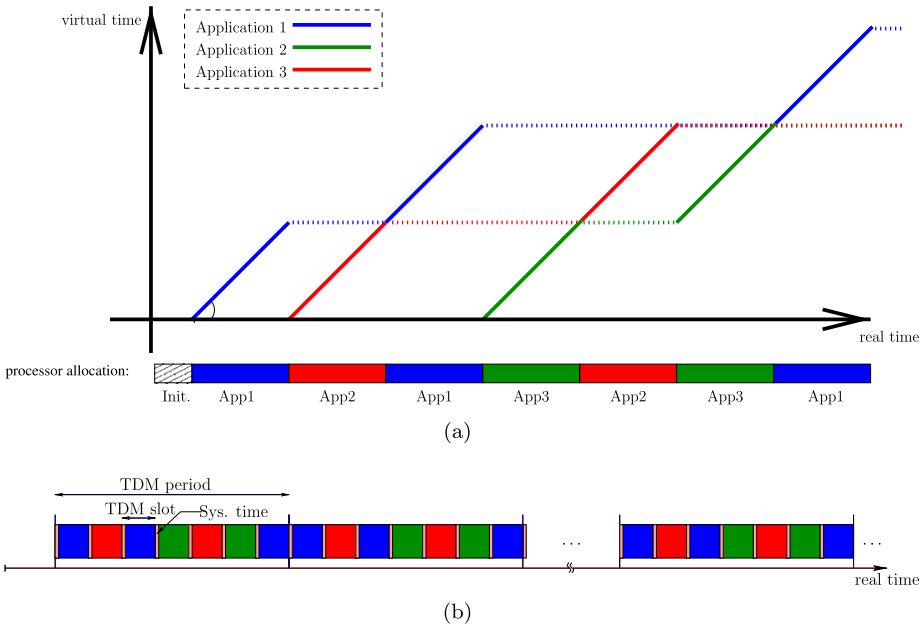


Fig. 4. (a) Processor allocation to multiple applications, and their virtual execution time vs. the system's real time. (b) The processor time allocated to the applications according to a TDM policy.

The possibility of performing task scheduling in application or system time results in different options of implementing the models of computation, i.e., KPN and dataflow, with the model of execution. Using the regular expression language, the sequence of operations for executing applications modeled in different models of computation is presented in Table 2. For KPN and dataflow we detail the cases in which the task (intra-application) scheduler is performed in: (i) system time, and (ii) application virtual time. The underlined operations in the table indicate the body of tasks, i.e., KPN processes or dataflow actors.

The execution model of KPN with tasks scheduled in system time, KPN (i), follows directly from the models of the task and the task scheduler. In this case in each system time the application and task are selected, $S_A S_T$, and in each application slot the task is executed, $\left(C + F_C^{[1,\infty)} F_A^{[0,1]}\right)^{[0,M]}$.

The execution model for KPN with tasks scheduled in the application time, KPN (ii), is more complex. In the application time, task selection has to be repeated whenever a task has finished, or, the task is either blocked or preempted. In detail, after a task is initially selected, i.e., S_T in Table 2, the task may `compute`, C, or `read` or `write`. In case of `read` or `write`, the FIFO has to be first checked, F_C. If the check fails, the current task is blocked, thus instead of polling for FIFO data or space, another task is selected, i.e., it starts from the beginning by executing another S_T. Otherwise, the check returns successfully and the FIFO buffer is accessed, F_A. After this access, another FIFO may be read or written, thus the procedure may be repeated. Furthermore, after a task iteration finishes or the task is preempted, another task is selected according the same algorithm above.

The execution model for dataflow is a composition of $F_A^{[0,N]} CF_A^{[0,N]}$, as the task body, and $(S_T F_C^{[0,N]})^{[0,N]}$, as the task scheduling, The latter may be placed in system time, Dataflow (i), or in application time, Dataflow (ii). If the task scheduler cannot find an eligible task, in Dataflow (i), it schedules the idle task, $(F_A^{[0,N]} CF_A^{[0,N]})^0$, and in Dataflow (ii), it continues polling for an eligible task, $(S_T F_C^{[0,N]})^{[0,N]} (F_A^{[0,N]} CF_A^{[0,N]})^0$.

The different options in implementation of the models of computation with model of execution results in different design choices for applications designers. In the next section we discuss the trade-offs in choosing each option, and how an application may benefit from each.

Table 2
Implementation of the models of computation with the unified model of execution when task scheduling is either in (i) system or (ii) application time.

Model of computation	System time	Application virtual time
NLP	S_A	C
KPN (i)	$S_A S_T$	$\left(\underline{C + F_C^{[1,\infty)} F_A^{[0,1]}}\right)^{[1,M]}$
KPN (ii)	S_A	$\left(\left(S_T \underline{(C + F_C^{[1,\infty)} F_A^{[0,1]})}\right)^{[0,M]}\right)^{[0,\infty)}$
Dataflow (i)	$S_A (S_T F_C^{[0,N]})^{[1,N]}$	$(\underline{F_A^{[0,N]} CF_A^{[0,N]}})^{[0,1]}$
Dataflow (ii)	S_A	$\left((S_T F_C^{[0,N]})^{[1,N]} \underline{(F_A^{[0,N]} CF_A^{[0,N]})^{[0,1]}}\right)^{[0,\infty)}$

5. Trade-offs in implementation of models of computation with the model of execution

A designer has many choices to implement an application on a platform. Two important ones are: (i) which model of computation to use and (ii) where to execute the task scheduling. Table 3 sum-

Table 3
Trade-offs summary.

Model of computation	System-time scheduling	Application-time scheduling
NLP	FRT, SRT, NRT	FRT, SRT, NRT
KPN	SRT, NRT	SRT, NRT
	variable status during an iteration	variable status during an iteration
	preemptive	cooperative
	non-work conserving	work conserving
	strictly verified scheduler	any scheduler
CSDF	FRT, SRT, NRT	FRT, SRT, NRT
	preemptive	cooperative
	constant status during an iteration	constant status during an iteration
	non-work conserving	work conserving
	strictly verified scheduler	any scheduler

marizes the trade-offs between the possible combinations of these two choices for NLP, KPN and dataflow models.

Some dataflow variants (e.g., CSDF) are analyzable, since the status of a task is *constant during an iteration* and it can be determined by checking the firing rules. This constant status is also visible in the implementation of the dataflow with the model of execution in Table 2, where no F_C operation exists in a dataflow task's body to cause unpredictable blocking time on a communication. Therefore, the operations in the dataflow task body and scheduler execute for a bounded number of repetitions ($[0, N]$), and provided that each operation finishes in bounded time, the model of execution corresponding to dataflow is predictable and subsequently amenable to temporal analysis.

In KPN implementation, an infinite number of F_C executions is possible ($[0, \infty]$), as presented in Table 2, which make the exact timing analysis of the KPN impossible. Thus KPN does not suit FRT applications, but only SRT and NRT applications. However, KPN can model dynamic behavior of applications, since the order of `read`s and `write`s and the number of tokens accessed are arbitrary. Moreover the status of a KPN task is not available before giving the control to that task, leaving room for less scheduling optimizations.

In case the task scheduler is executed in the system time, all scheduling decisions are taken exclusively in this time. When a task finishes or it is blocked before its slot ends, the remaining time is wasted. Thus this approach is *non-work-conserving*, potentially leading to a lower processor utilization. When executed in the application time, a new task may be scheduled immediately after a blocked or finished task. Therefore the entire application time can be utilized, i.e., this method is *work-conserving*. Moreover, a task scheduling policy is supported only under the condition that it is thoroughly verified and characterized, as the worst case execution time of the operations in system time should be tightly bounded. While this is a requirement for a real-time application, it is not necessary for non-real-time applications, where it can limit the available options. The limitation of task scheduling in application time on our current platform is that applications do not have access to timers and interrupts, unless these are virtualized. Hence on a platform that does not offer such virtual resources the scheduler policy has to be *cooperative*, i.e., cannot preempt tasks.

6. Implementation of the model of execution

Our model of execution is implemented on a composable MPSoC platform, in a form of software wrapper. In this section, we first describe the principles and basic building blocks of the existing target platform. Following that, we present the details of the additions that we made to this platform such that it implements the model of execution.

6.1. Background

Our target MPSoC platform consists of a hardware and a software infrastructure. Basically, the composability is a property of the platform, and the hardware and the software infrastructure implement this property. Here, we present the existing *composable* hardware and software platform [11,17] on top of which we propose the implementation that supports our model of execution.

6.1.1. Hardware infrastructure

The hardware infrastructure comprises processor and memory tiles interconnected via a Network-on-Chip (NoC) [41]. Fig. 5 shows an architecture template of the hardware infrastructure. A processor tile consists of a processor, a timer, local memory, shared memory, and Direct Memory Access (DMA) modules [42]. The

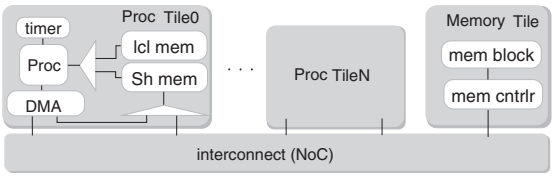


Fig. 5. Tiled architecture of a composable MPSoC platform.

timer always run at the maximum possible frequency of the system and keeps track of the system time as the reference of the real, physical time for the applications. A memory tile consists of a memory controller and a memory block.

Ideally the instructions and data of each task executing on a tile should reside in the local memory of the tile to minimize the data access latency. FIFOs are implemented in software and memory mapped. When the tasks of one application are mapped on different processor tiles, for inter-processor communication, the DMAs are used by the low-level API provided in native communication library of the RTOS to access remote memories (to either another tile, or an individual memory tile) [40]. This is implemented inside the model of execution, hence hidden from the applications. As we will see later in this section, the application designer operates with FIFO identification numbers when implementing inter-task communication. The mapping of the FIFO on memory and/or interconnect is not visible at this level.

All the shared resources are designed in such a way that they are virtualized to achieve application isolation and therefore system composability. The DMA and memory controller prevent a memory access of one application to block the processor and memory for an unknown time, and consequently violate the composability. The timer is the only one that can issue an interrupt to the processor in order to implement the fixed time slot durations. The NoC provides predictable and composable communication mechanism for the applications traffic, where the physical communication links are shared between them.

6.1.2. Software infrastructure

Following the asymmetric multiprocessing (AMP) method [16], an instance of a real-time operating system (RTOS) runs on each processor tile, as illustrated in Fig. 6. The main components of the RTOS are data structures, inter- and intra application schedulers, and the native communication libraries as presented in Fig. 6. The RTOS manages the data structure of the applications and their tasks in two types of control blocks, namely (i) *application control block* (ACBs), and (ii) *task control blocks* (TCBs). The control blocks are created and initialized before the applications start executing on the processor.

The RTOS provides an interface to the MPSoC resources, meaning that (i) it implements the virtual time of the application on top of the system's real time, i.e., it keeps track of the system time and application time slots of our model of execution, (ii) it schedules applications (and tasks) on the processor, and (iii) it offers an Application Programming Interface (API) that implements the low-level operations and facilitates the model of execution to access the MPSoC's resources, e.g., communication channels, memories, peripherals.

To implement the processor's TDM, the RTOS programs the timer to issue an interrupt in intervals of application slot's duration. The system time between two consecutive application slots, is known as "OS slot" in which the RTOS performs context switching and application scheduling, and it also monitors the applications' behavior, handles timer interrupts, and may execute task scheduling. The detailed operational time-line of the RTOS is presented in Fig. 7.

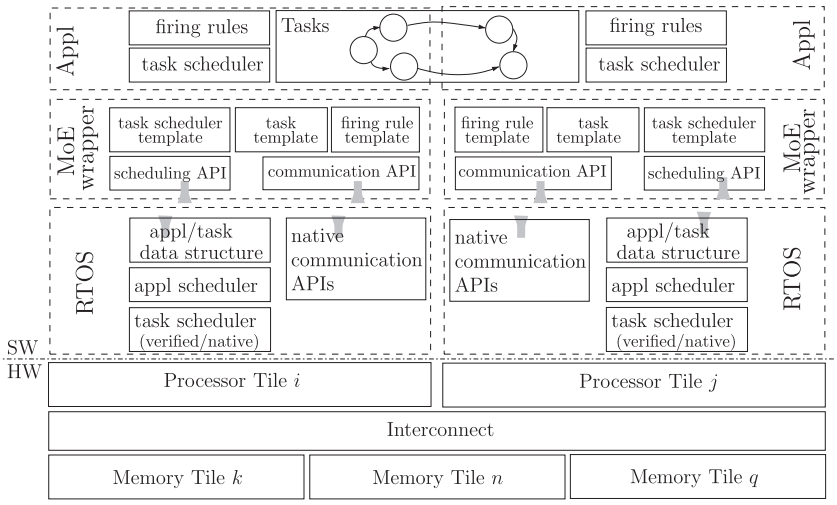


Fig. 6. The architecture of the MPSoC platform, with details of the software infrastructure.

The two hierarchical levels of scheduling, i.e., inter-application and intra-application (task) scheduling, are implemented each in different RTOS components, presented in Fig. 6. The RTOS provides only a safe API to the task scheduler implemented in an application, and totally isolates the scheduler from the underlying hardware. Consequently, the task scheduler cannot interface with the hardware interrupts, and therefore, it cannot implement preemptive scheduling policies. The low-level API is as follows:

(i) `os_get_current_task ( )` to get the ID of the current task.
(ii) `os_set_next_task (task_id)` to set the next task of the current application to be scheduled.

The native communication API directly accesses the communication resources of the tile, i.e., local memory, shared memory, and DMA. They internally implement software FIFOs with *data buffers* to store a number of tokens and *read* and *write* counters, and it keeps this administration data structure of each FIFO in a *FIFO control block* (FCB). The FCBs are managed according to the C-HEAP protocol [44] based on which the RTOS implements two FIFO operation primitives, as follows.

(i) `claim_fifo_data/space ( )` to acquire a data/space available in a FIFO.
(ii) `release_fifo_space/data ( )` to release the space/data of already consumed tokens.

Here we only discussed the API that is important for implementation of our model of execution, and the rest of the API are similar with what a conventional RTOS would offer. For more detailed discussion on the base-structure of the RTOS realization, we refer to [17].

6.1.3. Applications mapping

In the RTOS terminology, an *application* consists of a set of *tasks* that execute an infinite number of iterations, communicating via FIFOs. These tasks realize the body of NLP threads, KPN processes and dataflow actors in our model of execution. The given real-time streaming applications are parallelized manually or automatically using the existing tools such as PNGen [12] and Compaan [13] into a number of concurrent tasks implemented with either KPN or CSDF models.

To ensure that the required performance and timing constraints of an FRT application are guaranteed, the application has to be analyzed at design time. For this purpose, a complete analyzable model of the whole system is required. This system model integrates the time models of all the resources, e.g., DMAs and interconnect, used by the application, with the model of computation that implements the application [14]. The system model is then analyzed manually or automatically using the existing tools, such as SDF3 [15]. The analysis results may be used to (i) generate different partition and static mapping of the applications on the target MPSoC, and (ii) properly size the FIFOs for deadlock free execution and a given target performance of application.

In summary, in this work, we assume that a given real-time streaming application is implemented with KPN or CSDF models of computation and partitioned into a set of tasks which are mapped on one or more processor tiles. This process can be done manually or automatically, using existing tools. The focus of our work is then to provide an interface for the realization of the model of execution in between the existing RTOS and the given mapping of the applications, as illustrated in Fig. 2.

6.2. Implementation

Our unified model of execution is implemented on top of the existing RTOS layer of the software infrastructure. On every processor tile, the implementation of the model of execution is a software wrapper that consists of high-level scheduling and communication API and programming templates, as shown in Fig. 6.

In the rest of this section, we first describe the high-level API. The API is directly used by the applications and they abstract the

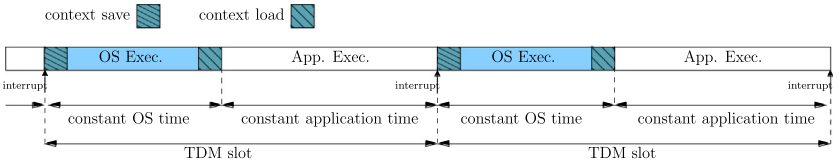


Fig. 7. A composable RTOS operational time-line.

low-level execution primitives of the RTOS; for example, a unified API is provided for the inter-task communication regardless of the mapping of tasks. Second, we present a unified programming template for the tasks, firing rules, and task schedulers. To implement the applications given in NLP, KPN, and CSDF models of computation, developers has to format them in these templates.

6.2.1. Application programming interfaces

In this section, we first discuss the scheduling operations and then the communication operations, for the KPN and the dataflow models of computation.

The scheduling operation in the KPN model boils down to selecting the next tasks according to a given policy, e.g., round robin. For this purpose, the low-level API described earlier in this section are exposed to the applications by the model of execution.

In the case of the dataflow model, the scheduling operation does not only select a task but also checks if the firing rules are met. The model of execution provides high-level API to express and check firing rules. To express a firing rule of a task, the application has to be able to set the number of tokens, known as *rate*, required for every iteration. The following API is proposed to set the consumption and production rate of a task's FIFOs:

(i) `os_set_fifo_consumption_rate (fifo_id, rate)`
(ii) `os_set_fifo_production_rate (fifo_id, rate)`

To check the firing rules of a task, the following API is proposed:

(i) `os_check_firing_rules (task_id)`

Internally, the function above goes over all the FIFOs associated to a task and, using the lower level API introduced earlier in this section, checks for required data/space in the FIFOs.

A KPN task may communicate with other tasks during its execution, hence it accesses a variable number of tokens from different FIFOs. For this purpose, the following API is proposed:

(i) `os_get_fifo_data (fifo_id, number_of_tokens, data_in)`
(ii) `os_set_fifo_data (fifo_id, number_of_tokens, data_out)`

Internally, the functions above utilize the native communication library of the RTOS to read/consume and write/produce data tokens from and to a FIFO. The communication API may be called anywhere in a KPN task body and the calls are blocking. In case that tasks are scheduled in application-time, when a task is blocked at a FIFO read/write, the task scheduler is immediately called and the current task is preempted.

A dataflow task (actor) requires a different communication scheme. First, all the task's FIFOs are checked (according to the firing rule), then if the communication is resolved, i.e., sufficient data and space are available, the task may start. We propose to wrap a dataflow task in a layer that first implements the FIFO check and access using low-level communication API then it calls the task computation. This wrapper is detailed in the next section. As a result, the task does not directly see its FIFOs; it is passed the data and buffer address that is needs for the current iteration. After the task computation ends, the output tokens are released by the wrapper.

6.2.2. Programming templates

The model of execution proposes specific programming templates for developers to implement the task computation, task firing rules, and task scheduler. Here, we describe these templates for the CSDF, KPN, and NLP models of computation.

Fig. 8 presents the template for tasks' body. In case that the task implements a dataflow actor, the arguments are the pointers to the input data and output space tokens and the number of each iteration (cycle). Since NLP and KPN processes perform the FIFO access operations dynamically in the body of the tasks and there is no notion of cycle in these models, the token pointers and cycle number arguments are set to `NULL` and `0`, respectively. For examples, Figs. 9 and 10 show how the dataflow actor and KPN process, that are presented in Figs. 3c and d, are implemented by the task template. Note that FIFO identification numbers (ids) that are used inside the dataflow actor and the KPN process are the local ids of the FIFOs, and the RTOS is initialized properly by the designer in order to link the FIFOs' local ids with the appropriate FCBs according to the application task graph.

A dataflow task should be accompanied with a set of firing rules, that may change from one task iteration to the other. To implement a firing rules function we proposed the template presented in Fig. 11. The firing rules function of a tasks is called automatically by the model of execution wrapper after each iteration of the task. As a result, the firing rules are updated for the next task iteration.

A unified task scheduler template is proposed to implement the cooperative policies, e.g., round robin, regardless of the application's model of computation. The template is illustrated in

```
void moe_task_template (void*** token_in,
                        void*** token_out,
                        int cycle)
{
  /* Start of task body */

  ...

  /* End of task body */
}
```

Fig. 8. The unified task template for NLP, KPN, and dataflow.

```
void dataflow_actor (void*** token_in,
                     void*** token_out,
                     int cycle)
{
  /* note that the dataflow actor fires once when called */
  /*
   * token_in[n][m] : the pointer to the m'th input
   *                  token data from FIFO n,
   * token_out[n][m]: the pointer to the m'th output
   *                  token space of FIFO n,
   * cycle          : the current task cycle in
   *                  case of implementing dataflow
   *                  MoC actor.
   */
  /* Body of the actor */
  fifo_token_type x,y;
  fifo_token_type z1,z2;

  x = ((fifo_token_type*) token_in[0][0])->value;
  y = ((fifo_token_type*) token_in[0][1])->value;

  z1 = ((fifo_token_type*) token_out[0][0])->value;
  z2 = ((fifo_token_type*) token_out[0][1])->value;

  z1 = compute1(x);
  z2 = compute2(y);
}
```

Fig. 9. An example dataflow task implemented with task template of the model of execution.

```
void kpn_process (NULL, NULL, 0)
{
  /*  process with finite token production
   *  and consumption
   */
  int i;
  for ( i=0; i< 100; i++) {
    fifo_token_type x,y;
    fifo_token_type z;
    os_get_fifo_data(fifo_id_1, 1, x);
    x = compute1(x);
    if ( x ==0 ) {
      os_os_get_fifo_data(fifo_id_2, 1, y);
    } else {
      y = compute2(x);
    }
    z = compute3(y);
    os_set_fifo_data(fifo_id_3, 1, z);
  }
}
```

Fig. 10. An example KPN task implemented with task template of the model of execution.

```
void task_firing_rule_template (int* consumer_fifo_rate,
                                int* producer_fifo_rate,
                                int  cycle)
{
  /* To implement firing rules for a dataflow actor */
  ...
}
```

Fig. 11. The unified firing rule implementation template for dataflow actors.

```
void task_scheduler_template (void* scheduler_arg)
{
  /* To implement a task scheduling policy */
  ....
}
```

Fig. 12. The unified task scheduler template.

Fig. 12. The required parameters for different scheduling policies may be set by the application designer and are passed by the RTOS to the scheduler via `scheduler_arg`. The application designer may use the scheduling API to implement any desired policy.

The implementations of the templates for tasks, firing rules, and schedulers, should be called by the model of execution according to its representations in Table 2 to realize the dataflow actors and KPN processes. For this purpose, the templates are wrapped in different set of sequential operations for each model of computation. Fig. 13 presents the model of execution wrapper for the dataflow actors. Similarly, KPN processes (or NLP) are wrapped as presented in Fig. 14, where there is no explicit native FIFO operation. Note that in case of application-time scheduling the task scheduler is called inside the kpn_process, or more precisely inside `os_get/set_fifo_data`, if a FIFO access is blocked.

```
void moe_dataflow_actor_wrapper (TCB* task){

  claim_fifo_data(task->token_in);
  claim_fifo_space(task->token_out);

  task->dataflow_actor(task->token_in,
                       task->token_out,
                       task->cycle);
  release_fifo_space(task->token_in);
  release_fifo_data(task->token_out);

  if ( application_time_scheduling ) {
    task_scheduler(task->scheduler_arg);
    context_switch();
  } else {
    wait_for_interrupt();
  }
}
```

Fig. 13. The dataflow task wrapper implementing the model of execution.

```
void moe_kpn_process_wrapper (TCB* task){

  task->kpn_process(NULL, NULL, 0);

  if ( application_time_scheduling ) {
    task_scheduler(task->scheduler_arg);
    context_switch();
  } else {
    wait_for_interrupt();
  }
}
```

Fig. 14. The KPN (NLP) task wrapper implementing the model of execution.

7. Case study

In this section we present experiments that execute multiple applications on our target MPSoC platform for two purposes: (i) to investigate the performance of applications implemented with different models of computation, and (ii) to study the system composability. The first investigation provides insights to the performance trade-offs (as in Table 3) in executing the different representations of one model of computation with our model of execution, as presented in Table 2. The second study justifies that the implementation of our model of execution does not violates the system composability.

The target platform has two processing tiles and one memory tile, communicating via an on-chip interconnect. The MPSoC is implemented on a Virtex 6 FPGA, available on Xilinx ML605 emulation board. For our experiments we set-up different usecases. A usecase is a number of applications executing concurrently the MPSoC platform. Here, we assume the applications are already parallelized, and the static mapping of tasks on specific processor tiles are given. The applications that we use are as follows: (i) a simple and (ii) a complex synthetic application, (iii) H.246 video decoder, and (iv) JPEG decoder. The simple synthetic application and the

JPEG are modeled with dataflow. The complex synthetic application is modeled with KPN and dataflow, and the H.264 is modeled with all models of computation, i.e., NLP, KPN and dataflow. For each usecase we implement two different *scenarios*, (i) OS-time task scheduling, and (ii) application-time task scheduling. Here, we consider the finishing time of an application as the metric for the application's performance. In every scenario, we make sure that application execution always starts at the same point in time. Therefore, smaller finishing time indicates better performance. In our experiments, we use the application performance as the criteria to study an application's execution behavior when it is implemented with different models of computation in each scenario.

7.1. Performance

In this section, we present the results of executing applications in different usecases on the MPSoC platform for the following two purposes, (i) to demonstrate how the implementations of the model of execution in Table 2 actually execute on the platform, and (ii) to give a real example of the performance trade-offs in implementing different models of computation with the model of execution, as discussed in Section 5.

For the first purpose we build a simple synthetic application that allows us to gain basic insights into how the model of execution behaves. Here we exemplify the application performance along the following design options: (i) OS- versus application-time scheduling, (ii) the processor belongs exclusively to the application versus the processor is shared, (iii) a sequential versus a parallel application execution. The last two are general points of interest for application design in any MPSoC platform, and the first is a degree of freedom existing in our platform.

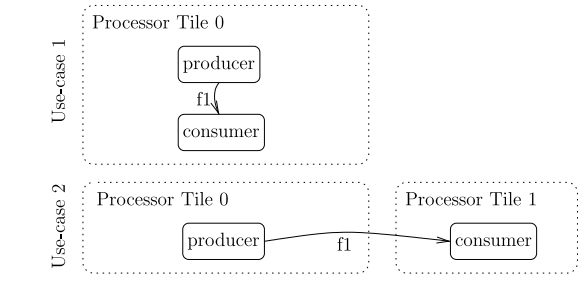
The simple synthetic application is composed of two tasks, i.e., a producer and a consumer, communicating via a FIFO. We implemented the dataflow model of this application. We set-up two usecases, as illustrated in Fig. 15a. In the first usecase the application is mapped on one processing tile, and in the second usecase, the producer task is mapped on processor tile 0 and the consumer task is mapped on tile 2.

Fig. 15b and c present the performance of the application in the two usecases, when the application's share of the processor is changed from 100% to 50%. For each set-up, we perform the experiments once with OS-time task scheduling and once with application-time task scheduling. Here, the processor time slots allocated to the applications are set to 100 k cycles. In the experiment, we investigate a range of task computation workloads.

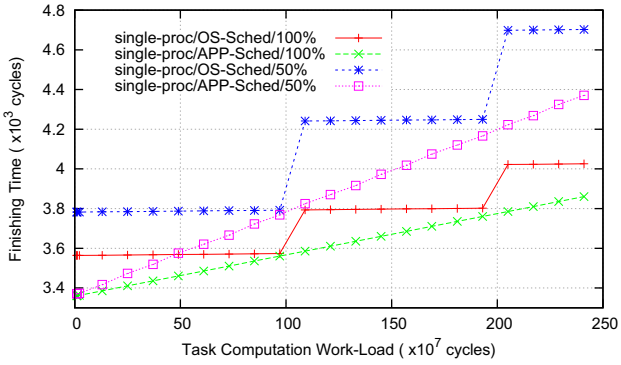
In both usecases, when the task finishes before the application slot ends, the application performs better if the tasks are scheduled in application-time than in OS-time. This is due to the fact that when a task does not utilize the whole slot, in OS-time scheduling, the remaining application time is wasted. However, in application-time, the task scheduler immediately schedules the next task. The difference between the results of OS-time and application-time in the Fig. 15b and c for each scenario, shows the wasted application time by the OS-time scheduling.

Even though we expect that the communication overhead increases in usecase 2, comparing to usecase 1, the results in Fig. 15c indicate better application performance in usecase 2, in overall. The reason is that, in usecase 2 the application time is not shared between the tasks on each processor, and each task owns the whole application time exclusively.

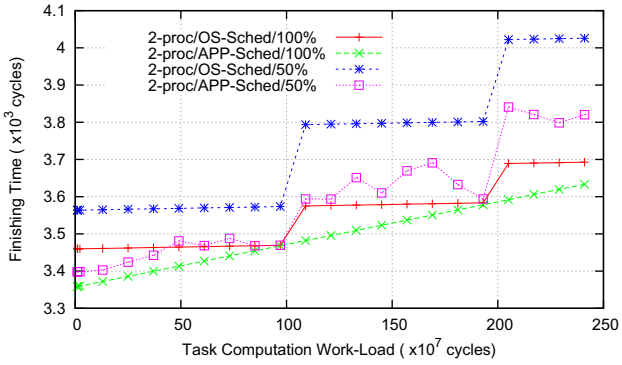
As it can be observed in Fig. 15c, the finishing time graph for the application-time scheduling scenario, when the application has 100% of the processor time, is linear, because, in this case, the finishing time is only dependent on the tasks' workloads which have been increased in regular steps of 10×10^7 cycles. Unlike this case, the finishing time graph is not perfectly linear, when the application has 50% of the processor time, because the waiting time caused by the fact that the application on the second tile is swapped out of the processor, can also affect the finishing time. In this case, the consumer task cannot start processing the data and it should wait until it gets the next application slot. The waiting time directly depends on how the TDM slots of an application on two processors are aligned. If the 50% slots allocated to the tasks on each processor are aligned perfectly, meaning that once the data is produced, the consumer reads it, the results would be perfectly linear as in the case of Fig. 15b. Otherwise, if data arrives on the other processor tile when the application has just lost the first processor, the consumer task should wait for the next coming slot to start processing. This is not the case when the application has 100% of the processor in which the tasks can start processing once data/space is available. This TDM misalignment effect also explains why in some cases, in Fig. 15c, we observe shorter finishing time for the case when the application has 50% of the processor, even if the task workload increases. When designing a real-time applica-



(a) Task graphs and mapping



(b) Usecase 1



(c) Usecase 2

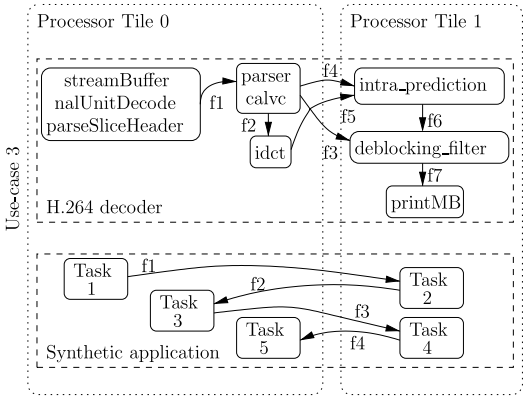
Fig. 15. (a) Two use case of a simple synthetic mapped on (i) one processor tile, and (ii) two processor tiles; finishing time for 10 iterations of (b) 1-tile use case, and (c) 2-tile use case.

tion, the worst-case TDM slots miss-alignment is taken into account to estimate the bounds on timing properties of the applications.

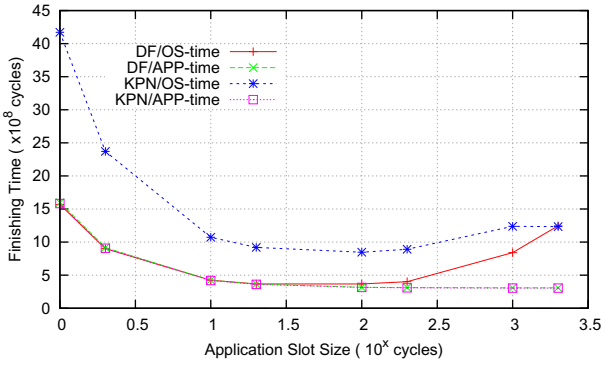
In order to investigate the trade-offs in implementing different models of computation with the model of execution (Table 2), we set-up a usecase which consists of two applications, namely, a synthetic one, and H.264 video decoder, as illustrated in Fig. 16a. The synthetic application is a five-task application implemented with KPN and dataflow models; each task has the same computation workload. The H.264 decoder is initially modeled in NLP, which is then parallelized in six tasks, in two models of computation, for KPN and for dataflow. We execute these applications concurrently using the round-robin task scheduling policy in OS-time and application-time. Here, we investigating the trade-offs involved in performance of the applications when an important system property, i.e., slot size, is changed. The results indicate how these design options would affect the applications performance.

The performance of the synthetic application is illustrated in Fig. 16b for various application slot sizes. Except for the small slot sizes, the task scheduling in application time leads to better performance, in both KPN and dataflow, because this implementation of scheduling is work-conservative and utilizes the entire application slot. Large application slot sizes leads to a large waste of application time in the case of OS-time scheduling. As expected, for small slot sizes the overhead caused by the frequent context switch between slots leads to poor performance in general. Here, for dataflow the performance differences between OS- and application-time scheduling are minor.

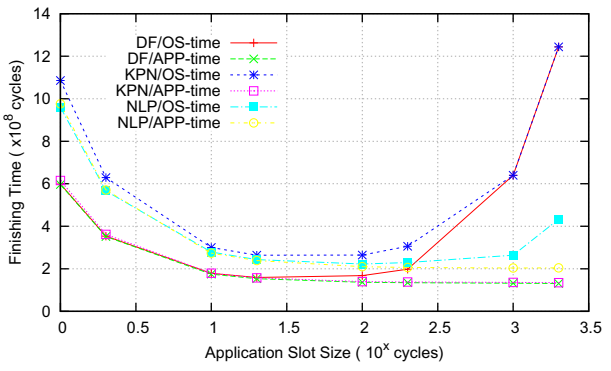
In the case of KPN, OS-time scheduling performs poorly regardless of the slot sizes, because the status of a task is not known when the task is selected and when the task is blocked the entire application slot is wasted.



(a) Task graphs and mapping

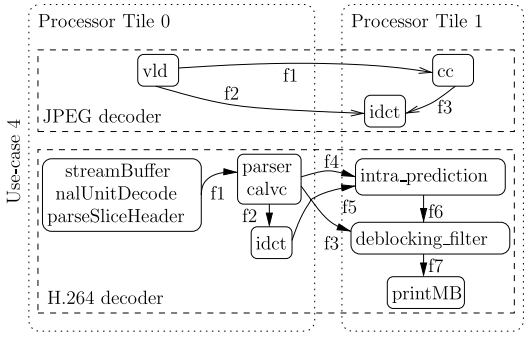


(b) The synthetic application

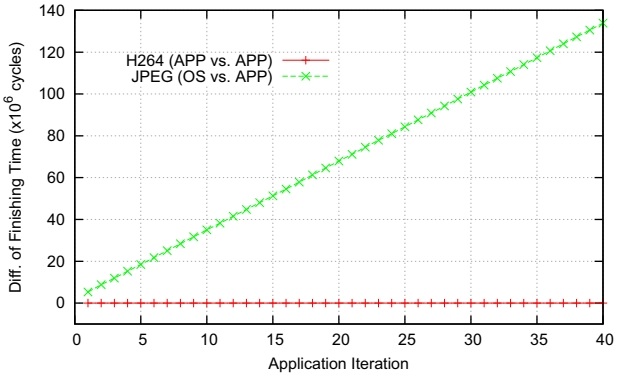


(c) H.264 video decoder

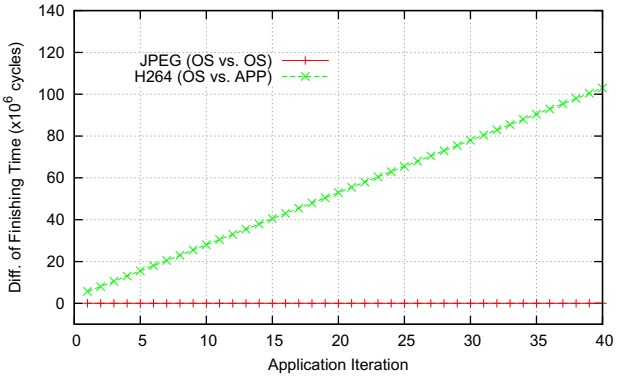
Fig. 16. (a) Synthetic and H.264 application on a 2-Tile MPSoC Platform; finishing time for 20 iterations of (b) the synthetic application, and (c) H.264 decoder.



(a) Task graphs and mapping



(b) JPEG task scheduling scenario is changed



(c) H264 task scheduling scenario is changed

Fig. 17. The finishing time difference between two execution scenarios of each application.

Fig. 16c illustrates the performance of the H.264 modeled in NLP, KPN and dataflow. NLP performs better than dataflow with OS-time scheduling for large application slot sizes, because the wasted time in each dataflow task exceeds the benefits of parallelizing the application. Similar to the synthetic application, in H.264 application-time scheduling leads to better performance, small slot sizes have large overhead, and the KPN with system time scheduling has the worst performance.

7.2. Composability

To verify the composability of the system when multiple applications execute on the platform, we set up a usecase with H.264 and JPEG applications as illustrated in Fig. 17a. We present two scenarios for this usecase when the task scheduling implementation is changed for one of the applications and for the other one is kept unchanged.

First, we execute the H.264 with the task scheduling in application-time, while the JPEG executes once with task scheduling in OS-time and another with application-time task scheduling. Fig. 17b presents the finishing time difference between the results of the experiments. The JPEG application shows no finishing time difference in two experiments, indicating that the timing properties of JPEG are unchanged, when the H.264 changes.

Second, we perform the two experiments while the H.264 executes once with task scheduling in OS-time and another with application-time task scheduling, for the case when the JPEG task scheduling is kept unchanged in OS-time. As expected, Fig. 17c presents the same results of no change in timing properties for the two executions of the H.264.

The experiments indicate that the timing properties of the applications are not affected when the behavior of other applications are modified, and therefore, the applications are independent. In other words, the system is composable.

8. Conclusions

In this paper we propose a unified model of execution to implement data-driven applications on a composable MPSoC platform. The applications can be realized in three different models of computation, e.g., Nested Loop Programmed (NLP), Kahn Process Network (KPN), and dataflow. The model of execution fills the gap between the MPSoC platform and the primitives of different models of computation. The execution model is formalized by, (i) defining the execution operations corresponding to the MoCs' primitives, (ii) proposing a time model of applications executing on a composable platform, and (iii) presenting different options of mapping the execution operations to the time model. The models of computation are mapped to the unified model of execution, and the trade-offs in different mapping options are discussed.

We implement the unified model of execution on top an existing composable MPSoC platform prototyped in FPGA. Using the proposed model, we experimentally investigate the applications performance and study the system composability by setting-up four usecases. In the first two usecases, we use a simple synthetic application, modeled in dataflow, to demonstrate how the implementations of the model of execution actually execute on the real platform. H.264 video decoder, modeled as NLP, KPN and dataflow, and a complex synthetic application, modeled as KPN and dataflow are used in the third usecase to give a real example of the performance trade-offs in implementing different models of computation with the model of execution. Finally, a usecase of H.264 and JPEG decoder demonstrates that the system composability holds true when a timing behavior of the applications are independent, at cycle level.

Acknowledgement

This work was partially funded by projects EU FP7 288008 T-CREST and 288248 Flextiles, Catrene CA104 Cobra and CA505 BENEFIC, CA703 OpenES, and NL STW 10346 NEST.

References

[1] W. Wolf, A.A. Jerraya, G. Martin, Multiprocessor system-on-chip (MPSoC) technology, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 27 (2008) 1701–1713.
[2] E. Lee, S. Neuendorffer, Concurrent models of computation for embedded software, in: IEEE Proceedings of Computers and Digital, Techniques, 2005.
[3] S. Stuijk, T. Basten, Analyzing concurrency in streaming applications, Journal of System Architecture 54 (2008) 124–144.
[4] E.A. Lee, A. Sangiovanni-vincentelli, A framework for comparing models of computation, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 17 (1998) 1217–1229.
[5] S. Stuijk, Predictable Mapping of Streaming Applications on Multiprocessors, Ph.D. thesis, Eindhoven University of Technology, The Netherlands, 2007.
[6] R. Turjan, B. Kienhuis, Translating affine nested-loop programs to process networks, in: Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), 2004.
[7] G. Kahn, The semantics of a simple language for parallel programming, in: Proceedings of Information Processing (IFIP) Congress, 1974. pp. 471–475.
[8] E.A. Lee, T. Parks, Dataflow process networks, in: Proceedings of the IEEE, 1995. pp. 773–799.
[9] J. Castrillon, R. Velasquez, A. Stulova, W. Sheng, J. Ceng, R. Leupers, G. Ascheid, H. Meyr, Trace-based KPN composability analysis for mapping simultaneous applications to MPSoC platforms, in: Proceedings of the Conference on Design, Automation and Test in Europe (DATE), 2010.
[10] H. Kopetz, Real-Time Systems: Design Principles for Distributed Embedded Applications, 1997.
[11] A. Hansson, K. Goossens, M. Bekooij, J. Huisken, CoMPSoC: a template for composable and predictable multi-processor system on chips, ACM Transactions on Design Automation of Electronic Systems (2009).
[12] S. Verdoolaege, H. Nikolov, T. Stefanov, PN: a tool for improved derivation of process networks, EURASIP Journal on Embedded Systems (2007).
[13] B. Kienhuis, E. Rijpkema, E. Deprettere, Compaan: deriving process networks from matlab for embedded signal processing architectures, in: Proceedings of the International Workshop on Hardware/software Codesign (CODES), 2000.
[14] A. Hansson, M.H. Wiggers, A.J.M. Moonen, K.G.W. Goossens, M.J.G. Bekooij, Enabling application-level performance guarantees in network-based systems on chip by applying dataflow analysis, IET Computers & Digital Techniques 3 (2009) 398–412.
[15] S. Stuijk, M. Geilen, T. Basten, SDF3: SDF for free, in: Proceedings of International Conference on Application of Concurrency to System Design (ACSD), 2006.
[16] G. Martin, Overview of the MPSoC design challenge, in: Proceedings of Design Automation Conference (DAC), 2006.
[17] A. Hansson, M. Ekerhult, A. Molnos, A. Milutinovic, A. Nelson, J. Ambrose, K. Goossens, Design and implementation of an operating system for composable processor sharing, Microprocessors and Microsystems 35 (2011) 246–260.
[18] D. Bui, E. Lee, I. Liu, H. Patel, J. Reineke, Temporal isolation on multiprocessing architectures, in: Proceedings of Design Automation Conference (DAC), 2011.
[19] A. Kumar, B. Mesman, B. Theelen, H. Corporaal, Y. Ha, Analyzing composability of applications on MPSoC platforms, Journal of Systems Architecture 54 (2008) 369–383.
[20] J.Y. Hur, T. Stefanov, S. Wong, S. Vassiliadis, Systematic customization of on-chip crossbar interconnects, in: Proceedings of the International Conference on Reconfigurable Computing: Architectures, Tools and Applications (ARC), 2007.
[21] J.Y. Hur, T. Stefanov, S. Wong, S. Vassiliadis, Customizing reconfigurable on-chip crossbar scheduler, in: Proceedings of IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP), 2007.
[22] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, E. Deprettere, System design using Kahn Process Networks: The Compaan/Laura approach, in: Proceedings of the Conference on Design, Automation and Test in Europe (DATE), 2004.
[23] C. Zissulescu, T. Stefanov, B. Kienhuis, E.F. Deprettere, Laura: Leiden architecture research and exploration tool, in: Proceedings of the International Conference on Field Programmable Logic and Applications (FPL), 2003.
[24] W. Haid, L. Schor, K. Huang, I. Bacivarov, L. Thiele, Efficient execution of kahn process networks on multi-processor systems using protothreads and windowed FIFOs, in: Proceedings of IEEE Symposium on Embedded Systems for Real-Time Multimedia (ESTIMedia), 2009.
[25] M. Dyer, M. Platzner, L. Thiele, Efficient execution of process networks on a reconfigurable hardware virtual machine, in: Proceedings of IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2004.
[26] I. Auge, F. Petrot, F. Donnet, P. Gomez, Platform-based design from parallel c specifications, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 24 (2005) 1811–1826.

[27] T.M. Parks, J.L. Pino, E.A. Lee, A comparison of synchronous and cycle-static dataflow, in: Proceedings of the Asilomar Conference on Signals, Systems, and Computers, 1995.
[28] M. Wiggers, M. Bekooij, P.G. Jansen, G.J.M. Smit, Efficient computation of buffer capacities for multi-rate real-time systems with back-pressure, in: Proceedings of International Conference on Hardware/Software Codesign and System, Synthesis (CODES+ISSS), 2006.
[29] M. Bekooij, M. Wiggers, J.L. van Meerbergen, Efficient buffer capacity and scheduler setting computation for soft real-time stream processing applications, in: Proceedings of International Workshop on Software and Compilers for Embedded Systems (SCOPES), 2007.
[30] O. Moreira, J.-D. Mol, M. Bekooij, J. van Meerbergen, Multiprocessor resource allocation for hard-real-time streaming with a dynamic job-mix, in: Proceeding of Real Time and Embedded Technology and Applications Symposium (RTAS), 2005.
[31] I. Sander, A. Jantsch, System modeling and transformational design refinement in ForSyDe, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 23 (2004) 17–32.
[32] J. Zou, S. Matic, E. Lee, T. Feng, P. Derler, Execution strategies for PTIDES, a programming model for distributed embedded systems, in: Proceedings of Real-Time and Embedded Technology and Applications Symposium (RTAS), 2009.
[33] T.A. Henzinger, B. Horowitz, C.M. Kirsch, Giotto: a time-triggered language for embedded programming, in: Proceedings of IEEE, 2001, pp. 166–184.
[34] J. Liu, E.A. Lee, Timed multitasking for real-time embedded software, IEEE Control Systems Magazine 23 (2003) 65–75.
[35] A. Ghosal, T.A. Henzinger, C.M. Kirsch, M.A.A. Sanvido, Event-driven programming with logical execution times, in: Proceedings of Hybrid Systems: Computation and Control (HSCC), 2004.
[36] V. Reyes, T. Bautista, G. Marrero, P. Carballo, W. Kruijtzer, CASSE: a system-level modeling and design-space exploration tool for multiprocessor systems-on-chip, in: Proceeding of Euromicro Symposium on Digital System Design (DSD), 2004.
[37] A.A. Jerraya, A. Bouchhima, F. Pétrot, Programming models and HW–SW interfaces abstraction for multi-processor SoC, in: Proceedings of the Annual Design Automation Conference (DAC), 2006.
[38] J. Eker, J.W. Janneck, J.W. Janneck, A structured description of dataflow actors and its application, 2003.
[39] A. Molnos, A. Beyranvand Nejad, B.T. Nguyen, S. Cotofana, K. Goossens, Decoupled inter- and intra-application scheduling for composable and robust embedded MPSoC platforms, in: Proceedings of International Workshop on Software and Compilers for Embedded Systems (SCOPES), 2012.
[40] B. Akesson, A.M. Molnos, A. Hansson, J. Ambrose Angelo, K. Goossens, Composability and predictability for independent application development, verification, and execution, in: Multiprocessor System-on-Chip, 2010, pp. 25–56.
[41] K. Goossens, A. Hansson, The Aethereal network on chip after ten years: goals, evolution, lessons, and future, in: Proceedings of the Design Automation Conference (DAC), 2010.
[42] B. Akesson, K. Goossens, Architectures and modeling of predictable memory controllers for improved system integration, in: Proceedings of Design, Automation Test in Europe Conference Exhibition (DATE).
[43] B. Akesson, A. Molnos, A. Hansson, J. Angelo, K. Goossens, Composability and Predictability for Independent Application Development, Verification, and Execution, Multiprocessor System-on-Chip: Hardware Design and Tool Integration (2010) 25.
[44] A. Nieuwland, J. Kang, O.P. Gangwal, R. Sethuraman, N. Busa, K. Goossens, R.P. Llopis, P. Lippens, C-heap: a heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems, Design Automation for Embedded Systems 7 (2002) 233–270.

Ashkan Beyranvand Nejad was born in Tehran, Iran, in 1983. He received the B.Sc. degree in electrical and electronic engineering from Iran University of Science and Technology, in 2005, and the M.Sc. degree in systems-on-chip design from Royal Institute of Technology (KTH), Stockholm, Sweden, in 2008. Since January 2009, he has been with the faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, The Netherlands, where he has persued his Ph.D. at Computer Engineering (CE) laboratory. His current research interests include composable, predictable embedded Systems-on-Chip, real-time systems, and execution models.

Anca Molnos was born in Fagaras, Romania in 1977. She received the M.S. degree in computer science from the "Politehnica" University of Bucharest, Romania and the Ph.D. degree in computer engineering from the Delft University of Technology, The Netherlands, in 2001 and 2009, respectively. The topic of her Ph.D. thesis was cache management for embedded multi-processor systems executing multimedia applications. Between 2006 and 2009 she was a senior scientist at NXP Semiconductors, The Netherlands, working on low power multiprocessors and distributed real-time systems. From 2009 to 2012 she was a Post-Doctoral researcher with the Delft University of Technology, The Netherlands. Currently she is a research engineer at CEA-Leti, France. Her research interests include composable, predictable multi-core embedded platforms and embedded multi-core resource management for lowpower, quality of service, and dependability.

Kees Goossens received his Ph.D. from the University of Edinburgh in 1993 on hardware verification using embeddings of formal semantics of hardware description languages in proof systems. He worked for Philips/NXP Research from 1995 to 2010 on networks on chip for consumer electronics, where real-time performance, predictability, and costs are major constraints. He was part-time full professor at the Delft university of technology from 2007 to 2010, and is currently full professor at the Eindhoven university of technology, where his research focusses on composable (virtualised), predictable (real-time), low-power embedded systems.