Composable Virtual Platforms for Mixed-Criticality Embedded Systems

Ashkan Beyranvand Nejad

Composable Virtual Platforms for Mixed-Criticality Embedded Systems

Ashkan Beyranvand Nejad

Composable Virtual Platforms for Mixed-Criticality Embedded Systems

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen

op woensdag 5 november 2014 om 12:30 uur

door

Ashkan BEYRANVAND NEJAD

Master of Science in Systems-on-Chip Design
Kungliga Tekniska Högskola (KTH), Zweden
geboren te Teheran, Iran

Dit proefschrift is goedgekeurd door de promotor:
Prof. dr. K.G.W. Goossens

Copromotor: Dr. Anca M. Molnos

Samenstelling promotiecommissie:

Rector Magnificus	voorzitter
Prof. dr. Kees G.W. Goossens	Technische Universiteit Delft, promotor
Dr. Anca M. Molnos	CEA LETI, copromotor
Prof. dr. Koen L.M. Bertels	Technische Universiteit Delft
Prof. dr. Ben Juurlink	Technische Universität Berlin
Prof. dr. Henk Corporaal	Technische Universiteit Eindhoven
Dr. Sorin D. Cotofana	Technische Universiteit Delft
Prof. dr. Piet F.A. Van Mieghem	Technische Universiteit Delft, reservelid

Ashkan Beyranvand Nejad
Composable Virtual Platforms for Mixed-Criticality Embedded Systems

Met samenvatting in het Nederlands.

ISBN 978-94-6186-376-8

Cover design: The cover is designed by the author using the *word cloud* as the visual representation of the Introduction chapter of this dissertation, where the frequency of appearing each word in the text is shown relatively with its font size. The cloud is created using *Wordle™* web-based tool.

Printed in The Netherlands

Dedicated to my dear parents

and

to the love of my life, Golnoosh.

Abstract

RECENT trends show a steady increase towards concurrently executing more and more applications on a single embedded system. Multi-Processor System-on-Chip (MPSoC) architectures are proposed to allow complex design of embedded systems. This is achieved by integrating as many processing resources as possible on a single chip and therefore enabling the execution of multiple applications on a single embedded System-on-Chip (SoC). Due to cost implications, the applications have to share some resources when executing concurrently on these systems. To fully exploit the computational power of an MPSoC, an application is further split into a number of concurrent tasks. Depending on the intrinsic behavior of an application, its tasks may be either data- or time-interdependent, and accordingly, two different models of computation, namely data-driven and time-driven, are used to implement such an application. Besides this, applications typically have timing requirements expressed in three categories of firm, soft, and non real-time requirements. In a *mixed time-criticality* system, the applications that execute concurrently on a single embedded platform have widely varying real-time requirements, where resource sharing causes interference between the applications. In order to execute real-time applications on an embedded platform, the system has to be predictable to ensure that the timing requirements of the applications are met. Moreover, to enable independent design, verification, and integration of mixed-criticality applications, the system has to be strongly composable, i.e., concurrently executing applications are temporally isolated in such a way that the actual-case temporal behavior of each application is unaffected at the cycle-level. In this way, temporal interference between the applications is completely prevented.

In this dissertation, we address two main challenges in designing and prototyping mixed time-criticality systems: (i) realizing strongly composable Virtual Platforms (VPs) for mixed-criticality embedded systems, and (ii) proposing a uniform abstract execution layer for applications expressed with different models of computation. Here, we target time-driven models of computation and, Kahn Process Network (KPN) and dataflow (specifically, Cyclo-Static DataFlow (CSDF)) as the two variants of data-driven models of computation. On the basis of these challenges we answer the main research question of *how to design and execute multiple applications concurrently on an embedded system, given that the applications are realized with different models of computation and having different levels of time-criticality?*

This dissertation proposes a solution to create a VP for every application by virtual-

izing all the hardware resources that are involved in the execution of the applications. For this, a temporal partitioning technique is applied to the CompSOC hardware architecture. On top of this architecture, the CoMik microkernel is designed as a minimum privileged software layer to provide resource partitioning. CoMik creates, controls and schedules processor partitions, and executes an application in its partition by virtualizing the processor's software hooks, offering an Application Programming Interface (API) to each application to use its allocated resources. Applications can therefore execute directly on their virtual platforms in the same manner as on a dedicated hardware platform. However there exists an execution abstraction gap between the models of computation's semantics and the platform's primitive operations. To fill this gap, a model of execution is proposed to define a common set of execution operations and their orchestrations in order to implement an specific model of computation.

The model of execution is implemented in the form of a lightweight operating system library, namely CompOSe, which is instantiated in every partition of a VP. CompOSe is designed in a number of software units and is implemented in such a way that it does not introduce any unpredictability in executing an application and complies with the composability property of the system provided by the CoMik microkernel.

We demonstrate that our proposed technique enables concurrent composable, predictable execution of applications realized with multiple models of computation by using two experimentation setups. One, a Matlab simulation environment is used to investigate the temporal behavior of the CoMik microkernel. Two, an FPGA prototype of the CompSOC platform is used to study the composability property and support of multiple models of computation by the CompOSe Real-Time Operating System.

Acknowledgements

Well said by Ernest Hemingway, *"It is good to have an end to journey toward; but it is the journey that matters, in the end"*. My journey began from 1987 when I started learning to read and write from my grandmother RIP. It was two years later, 22nd of September 1989, the night before the day that I went to the first grade of school, my farther called me in to tell me something very important: *"Tomorrow you are going to take the first step of a long journey. Since you already know how to read and write, you are free to decide not to go to school. But, if you would decide to go, you should know that it could be a long journey that you cannot stop in the middle, and you have to go until the end. This is not what we force you to do, but the journey itself is so fascinating that you yourself cannot stop"*.

Now that you are reading this PhD thesis, I obviously decided that night to start the journey. I am now very satisfied with my decision, although I have gone through many ups and downs along the way to this point. Here, I cannot name all the people who helped and accompanied me in the first twenty years of this journey, before starting my PhD work, but I am thankful to all of them. I would also like to thank the committee members of my PhD defense and you, the reader, who may find this thesis interesting to you.

From the last six years, first and foremost I would like to sincerely thank my thesis promotor Prof.Kees Goossens for giving me the opportunity of pursuing my PhD under his supervision. I started my work with him from 2008 when I moved to the Netherlands for doing my M.Sc. thesis on a topic proposed by him at the time he was still in NXP Semiconductors. Since then, I have always enjoyed working with him and learnt a lot of things from all the technical and the non-technical discussions that I have had with him. I admire his high-quality research work and his personality, especially his patience.

I am sincerely grateful for the contribution of my thesis copromotor Dr.Anca Molnos to my work. Anca has been not only a good colleague for me who I enjoyed all the moments of working with her, but also a very nice friend of mine. I should acknowledge her noticeable influence on my writing skills for presenting a scientific work and her valuable feedback on the draft of this dissertation. I wish to express my appreciation and thanks for my great office-mate, good friend, and collaboration partner, Andrew Nelson. We have shared many memorable moments during the last five years in the trips, the meetings, the discussions, and most importantly in the work that we have done together. I would never forget his kind help during the preparation of my thesis, and I would also like to thank his girl-friend, Jorinde de Boer, for translating my propositions into Dutch.

From the CompSoC team, my thanks go to Radu Stefan, Benny Akesson, Martijn Koedam, Sven Goossens, Karthik Chandrasekar, Davit Mirzoyan, Manil Dev Gomony, and Shubhendu Sinha for all the great collaborations and discussions that we have had together. I should specifically thank Martijn and Sven for kindly having the abstract of my thesis translated into Dutch. I would also like to extend my gratitude to Bart Vermeulen who used to be the daily supervisor of my M.Sc. thesis and has been a good friend of mine for the last six years.

From the Computer Engineering Laboratory of TUDelft, my sincere appreciation goes to Prof.Koen Bertels for all his supports during the last five years. I would like to thank Lidwina Tromp, the secretary of the group, Erik de Vries and Eef Hartman, the IT administrators, for all their facilitating services kindly provided to us. I would also like to thank Arash Ostadzadeh, a good friend and colleague, for all the nice discussions that we had, and acknowledge his kindness for sharing this thesis template with me.

Last but the most important, I express my greatest gratitude towards my parents, Mehri and Reza, for all their encouragements and supports over the course of my life. Without always feeling them beside me, even from the long distance in the last eight years, I could not have taken even one step forward. I want to also thank my sister, Elham, for her unlimited supports and accompany in my life, and her husband, Babak, for his respectful encouragements from the time he joined our family. Finally, from the bottom of my heart, I would like to thank the love of my life, Golnoosh, for making my life sweat by being beside me from the first months of starting my PhD journey, and her endless supports without which I could not finish this dissertation.

Ashkan Beyranvand Nejad　　Delft, The Netherlands, November 2014

Table of contents

List of figures

Chapter 1 **1**

Chapter 2 **11**

Chapter 3 **29**

Chapter 4 **53**

Chapter 5 **75**

Appendix A **97**

List of tables

Chapter 2

List of listings

Chapter 2 **11**

Chapter 4 **53**

Appendix A **97**

List of Acronyms

ACB	Application Control Block	62
AMP	Asymmetric Multiprocessing	17
API	Application Programming Interface	94
ARINC	Aeronautical Radio Incorporated	50
AUTOSAR	AUTomotive Open System ARchitecture	50
CCB	CoMik Control Block	98
CCM	Clock Control Module	97
CSDF	Cyclo-Static DataFlow	21
DDR	Double Data Rate	14
DLMB	Data Local Memory Bus	16
DPLB	Data Processor Local Bus	15
DTL	Device Transaction Language	16
ECU	Electronic Control Unit	1
FCB	FIFO Control Block	63
FIFO	First-In-First-Out	22
FPGA	Field Programmable Gate Array	114
FRT	Firm Real-time	21
FSL	Fast Serial Link	16
ILMB	Instruction Local Memory Bus	14
IMA	Integrated Modular Avionics	1
KPN	Kahn Process Network	21
MMIO	Memory-Mapped Input/Output	15
MMU	Memory Management Unit	16
MPSoC	Multi-Processor System-on-Chip	93
NI	Network Interface	16
NOC	Network-On-Chip	16
NRT	Non Real-time	21

OEM	Original Equipment Manufacturer	1
OS	Operating System	94
PCB	Partition Control Block	95
PIT	Programmable Interrupt Timer	97
PTI	Programming Timer Interrupt	97
RISC	Reduced Instruction Set Computer	14
RDMA	Remote Direct Memory Access	14
RR	Round-Robin	58
RTOS	Real-Time Operating System	17
SC	Software Container	94
SDF	Static Dataflow	21
SDRAM	Synchronous Dynamic Random-Access Memory	14
SoC	System-on-Chip	93
SRT	Soft Real-time	21
TCB	Task Control Block	62
TDM	Time Division Multiplexing	93
TIFU	Timer-centric Interrupt and Frequency Unit	95
VM	Virtual Machine	29
VMM	Virtual Machine Monitor	29
VP	Virtual Platform	93

Terminology

In this dissertation, we refer to several terms that are ambiguous and may specifically cause confusion when used in the context of Computer Science, especially in the field of embedded systems. In the following, we clarify the most important and frequently used terms with references to the first place that each terms is defined and used in this dissertation.

Application is a set of algorithmic computational operations that realize a functionality which may be split is a set of communicating tasks [Section 2.5].

Task is a piece of sequential code that implements a part of an application's functionality [Section 2.4.2].

Model of Computation is a model that implements the application using a set of formal semantics defined to express the computational operations [Section 2.5]

Model of Execution is a model that defines a set of execution operations and their orchestrations in order to implement an specific model of computation. It has to explicitly define computation, communication, and scheduling operations for each individual model of computation [Section 4.1].

Node is a functional mapping from inputs to outputs in a model of computation [Section 2.5.1].

Process is a node in the KPN model of computation and consists of a sequence of `read`, `compute`, and `write` operations which may be interleaved in any order [Section 2.5.1].

Actor is a node in the CSDF model of computation with a sequence of `consume`, `compute`, and `produce` operations, in this strict order. 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 [Section 2.5.1].

Job is a node in the time-driven model of computation. It is ready to execute at an specific moment in time, when it assumes the data and the space that it may require for its execution are available [Section 2.5.2].

Time-Criticality is a level of timing requirements expressed in one of the three categories of firm, soft, and non real-time requirements [The introduction of Chapter 1].

Predictability is a system property which is defined as a level of how well the timing characteristics of the system are defined and implemented so that the system temporal behavior is known before it actually starts running the applications [Section 2.1].

Composability is the property that the temporal (and functional) behavior of an application is isolated and independent from the one of other concurrent applications. A system is either *weakly* or *strongly* composable [Section 2.2].

Virtualization is traditionally the technology to provide an illusion of execution resources to applications so that the applications behave like running directly on the bare resources [The introduction of Chapter 3].

(Processor tile) Partition is created by the CoMik microkernel on every processor tile and it includes a composable temporal processor partition, an spatial *dmem* partition, a number of dedicated Remote Direct Memory Access (RDMA) modules [Section 3.2].

Virtual Platform (VP) is logically defined as a set of resource partitions allocated to one application. Every VP provides the application with an illusion of a dedicated actual physical hardware architecture [Section 3.2].

Microkernel a minimum privileged software layer that is designed to provide the essential services of partitioning for the purpose of virtualization [Section 3.3].

CompSOC is a System-on-Chip (SoC) template developed following composability and predictability paradigm, and implements a tile-based multi-processor architecture which consists of a number of processor and memory tiles communicating via an on-chip interconnect [Section 2.3.1].

CoMik is a microkernel that, on each processor, creates, controls, and schedules a number of partitions each of which is allocated to one application [Section 3.3].

CompOSe is a Real-Time Operating System (RTOS) library that could execute as part of a *partition* created by the CoMik microkernel and implements the execution primitives proposed by the model of execution [Section 4.2].

Time-triggered scheduling is a scheduling category in which the scheduler is invoked by a timed event, e.g., timer interrupt [Section 2.4].

Event-triggered scheduling is a scheduling category in which the scheduler is invoked by any other event than a timed event, for example, an I/O interrupt or data availability [Section 2.4].

Preemptive scheduling is a scheduling class in which the invocation of the scheduler preempts the executing task [Section 2.4].

Cooperative scheduling is a scheduling class in which the executing task is allowed to finish (or signal that it can yield the processor) before the scheduler is invoked [Section 2.4].

CHAPTER 1

Introduction

N*OWADAYS* embedded systems are widely used computing platforms in various domains such as automotive, consumer electronics, medical devices, avionics, etc. The common increasing demand of all these domains is to execute more and more applications concurrently on a single embedded system platform. Applications are functionally-independent software units that are developed by possibly different parties. The primary reason for such a demand varies per application domain. For example, in automotive domain, there are more than 70 individual Electronic Control Units (ECUs) developed by various Original Equipment Manufacturers (OEMs) integrated and communicating inside a middle class car [13]. In this domain, reducing the number of ECUs and increasing the number of applications executing on each one, not only reduces the manufacturing cost but also improves the reliability and maintainability of the systems since, for instance, the cabling complexity is decreased [37]. However, in avionics domain, to some extend the cost of reducing the number of electronic computing hardware modules is less important than reducing the weight of the aircrafts and complexity of the systems is an important objective [46, 70].

Shrinking the feature size of the transistors has led to low-cost integration of more and more number of processing resources on a single chip and consequently allows designing complex embedded system platforms using Multi-Processor System-on-Chip (MPSoC) [44]. Such platforms enable executing multiple applications on a single embedded System-on-Chip (SoC) architecture by providing each application with the required resources. However, since the requirements of applications grow faster, these architectures are still resource constrained. For cost reasons, the applications executing on these platforms have to share some resources, such as processors, interconnect, memory blocks, etc. An example of this trend is in avionics domain where the traditional federated computing architectures in which each subsystem occupied a physically separate hardware component is replaced with integrated computing architectures following Integrated Modular Avionics (IMA) design concept, in which multiple applications share the common computing platform [70, 81].

Furthermore, belonging to various domains such as consumer electronics, automotive, avionics, etc., the applications realize different functionality with different intrinsic behaviors. For example, multimedia applications have different functional behavior

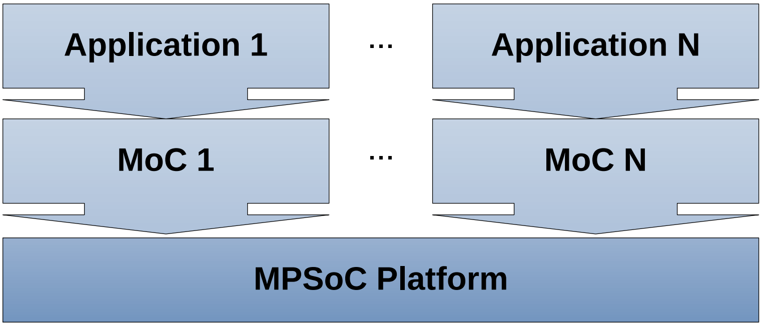


Figure 1.1: *An embedded system stack.*

compared to an engine control application. One example of multimedia applications is a video codec that typically receives an streaming input of data to which it applies some computation and outputs an stream of data for illustration. Instead, the engine control application, as an example of automotive applications, periodically reads some sensors, applies some computation, and produces a signal to control an actuator at specific moment in time. Thus, different models of computation are required to implement different applications. A model of computation implements the application using a set of formal semantics defined to express the computational operations [43]. Traditionally, the applications are first implemented in a sequential model of computation using an imperative programming language such as C. On MPSoC platforms multiple applications may execute in parallel. 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. Depending on the intrinsic behavior of an application, its tasks may synchronize naturally either on the basis of data availability or on specific times. In other words, the tasks may be either data- or time-interdependent. Accordingly, two different models of computation, namely data-driven and time-driven, are used to implement such inter-dependencies between the tasks.

Besides having different functional requirements, the embedded applications typically have timing requirements expressed in three categories of firm, soft, and non real-time requirements. These requirements are categorized with respect to the time, known as deadline, before which the application must produce an output and be finished. A Firm Real-time (FRT) application has strict timing requirements that must never miss a deadline otherwise damage may be caused to the system or environment [53]. While FRT applications can be found in various industries, the automotive and avionics applications are typically from such a kind. A Soft Real-time (SRT) application however may occasionally miss a deadline, whereas a Non Real-time (NRT) application do not have to meet any timing requirement and it has no deadline. The SRT and NRT applications exist in almost all the domains.

When the applications executing concurrently on a single embedded platform have different real-time requirements, the system has *mixed criticality* property. The term mixed-criticality may also apply to systems (mostly in automotive and avionics industries) with mixed-safety criticalities, however in this dissertation we aim at mixed time-criticality systems.

In summary, the ongoing trend of embedded systems is to design mixed time-criticality MPSoC platforms in order to execute multiple applications from various domains such as automotive, avionics, etc. The applications are implemented with either of data- or time-driven models of computation each of which has its own timing and functional requirements that has to be met when executing on the platform. Figure 1.1 illustrates an abstract overview of such system's stack. Now, let us explain the existing challenges in designing and prototyping such systems.

1.1 Challenges

To execute real-time applications on a platform, the system has to ensure that the timing requirements of the applications are met. For this purpose, the system has to be designed in such a way that the timing characteristics of the system are so well defined that the temporal behavior of the applications can be verified against their requirements before actually executing on the platform. Conventionally, this implies the *predictability* property of the system. Predictability is defined as a level of how well the timing characteristics of the system are defined and implemented so that the system temporal behavior is known before it actually starts running the applications.

In a mixed-criticality system in which multiple real-time applications execute concurrently on an embedded platform, resource sharing causes interference between the applications. Such interference might be either predictable (bounded) or unpredictable (unbounded). The predictable interference is typically between real-time applications with the bounded temporal execution behaviors; whereas the unpredictable interferences result from the non-real-time applications. In case of bounded unpredictable interference, as the number of concurrent execution combinations of the applications may grow exponentially, the verification and integration would become dramatically difficult [53]. It would be even worse when the applications are developed by different parties and all the applications has to be available at design time to be verified against their timing requirements. In the case of unbounded unpredictable interference, it is not possible at all to verify the timing requirements of the real-time applications and therefore to integrate them with the non-real-time ones.

In order to prevent the interference, the applications have to execute temporally isolated. For this, *composability* is proposed as one of the embedded system's properties. It is defined as: the temporal (and functional) behavior of an application is isolated and independent from of other concurrent applications [53, 85]. We denote a system as either *weakly* or *strongly* composable. Weak composability is denoted as a temporal-interference relaxed isolation between applications, where, with the help of predictability, a system could make the worst-case temporal behavior of an application unaffected by the other applications [55]. In this way, the system can guarantee a minimum service provided to the applications. However, in the mixed criticality systems, weak composability does not help to reduce the complexity of system verification and integration as it is not possible to come up with worst-case bounds for temporal behavior of the non real-time applications. *Strong* composability [4, 26] however is the isolation between the applications in such a way that the *actual-case temporal behavior* of each application is unaffected. This enables independent design, verification, and integration of mixed-criticality applications.

In order to implement a composable system, virtualization technology has gained recently a lot of attentions as a design trend in embedded systems [32]. In this technology an illusion of the execution resources are provided to the applications so that the applications behave like running on the bare hardware platform even though the resources are shared. Such an illusion of the system is called a *Virtual Platform (VP)*. Implementing the composability property for the system is then the challenge of creating temporally isolated VPs for the applications.

To virtualize the resources they have to be either dedicatedly allocated to an application or they have to be partitioned. Depending on the service provided by a resource, it can be partitioned either temporally or spatially [65]. For instance, in this context, two well-known standards, namely Aeronautical Radio Incorporated (ARINC) and AUTomotive Open System ARchitecture (AUTOSAR), have been developed in the automotive and the avionics domains, respectively. Real-Time Operating Systems (RTOSs) that are designed based on these standards are to support the partitioning of hardware resources in order to enable realization of VPs for mixed-criticality systems.

In the avionics domain, ARINC standard is a specification for time and space partitioning in mixed-criticality avionics RTOSs [83]. It specifies partitions at application level, where one or more applications with different criticality can belong to one partition [96]. The standard also specifies Application Programming Interfaces (APIs) for abstraction of the application from the underlying hardware and software platform. The RTOSs developed based on this standard guarantee the minimum amount of service that a partition receives. Thus, an application executing in a partition is affected by the presence/absence of other partitions.

In the automotive domain, AUTOSAR is a standard automotive software architecture, jointly developed by automobile manufacturers, suppliers and tool developers [9]. It contains an RTOS specification which defines real-time performance, scheduling strategy and temporal partitioning for executing applications with mixed-criticality. In this standard the partitions are implemented with a schedule table, and a time monitoring mechanism is used to limit the worst-case execution time of the applications. Thus, an application timing behavior can be possibly affected by the execution of other applications.

None of this existing systems fully comply with the definition of strong composability. As we argued, strong composability is very strict in the sense that the actual-case behavior of the applications has to be isolated. This means that the actual (and worst-case) timing properties of the an application are cycle-accurately independent of the other applications. A challenge here is to create VPs that comply with this strict definition of the composability.

Furthermore, the applications that are going to execute on VPs are possibly expressed with different models of computation, i.e., data- or time-driven. Each model of computation has its own execution primitives for computation and communication operations. Besides this, in the case of concurrent models of computation where an application is split into a number of tasks, scheduling operation comes into the application execution play. In order to hide the execution details of the implementation on the platform, these operations are typically specified at a high level of abstraction. For example, in a data-driven model of computation where inter-task communication is performed by means of First-In-First-Out (FIFO) queues, a read operation of a task is an abstraction of checking a queue for available data and retrieving the data from possibly a remote memory location on the platform. Thus, there is a gap of execution abstraction between the models of computation's semantics and the primitive execution operations supported by the platform. This gap causes a huge design effort to support the applications with various models of computation. A design challenge here is therefore to fill this gap by an intermediate execution layer to support the different models of computation in a unified manner.

In summary, in this dissertation we address two main design challenges: (i) realizing strongly composable VPs for mixed-criticality embedded systems, and (ii) proposing a uniform abstract execution layer for applications expressed with different models of computation on the VP. These two challenges lead us to the main research question of this dissertation that can be presented as follows:

How to design and execute multiple applications concurrently on an embedded system, given that the applications are realized with different models of computation and having different levels of time-criticality?

The rest of this chapter is going to explain our proposed solution to answer this question, where the solution makes several contributions to address the two aforementioned challenges.

1.2 Overview of the Solution

In order to tackle the introduced research question our approach is to follow the explained design trends in embedded systems where the virtualization technology is applied to make the mixed-criticality systems predictable and composable.

In this approach the aim is to create a VP for every application by virtualizing all the resources that are involved in the execution of the applications. For this, we use the partitioning technique. This technique is applied to different resources of the platform. Depending on the service type provided by a resource, it can be partitioned temporally or spatially [65]. In the context of mixed-time criticality systems, our main focus is on temporal partitioning of the resources. For this, we propose a complete platform-based approach as illustrated in Figure 1.2. This approach mitigates the complexity of a mixed-criticality system stack by proposing a multi-layered virtualization software platform on top of an MPSoC heterogeneous hardware architecture.

Starting from the bottom of the stack, in the hardware platform, each resource has to be specially designed and implemented for the purpose of guaranteeing predictable and composable execution of applications with mixed-time criticality. CompSOC is a SoC template developed following composability and predictability paradigm [29]. Here, we use this template as the basis of our hardware architecture. CompSOC has a tiled-based MPSoC architecture which consist of a number of processor and memory tiles communicating via a Network-On-Chip (NOC) [22, 27, 31, 89]. All its resources are designed and arbitrated between the executing applications in such a way that they realize predictable and composable system.

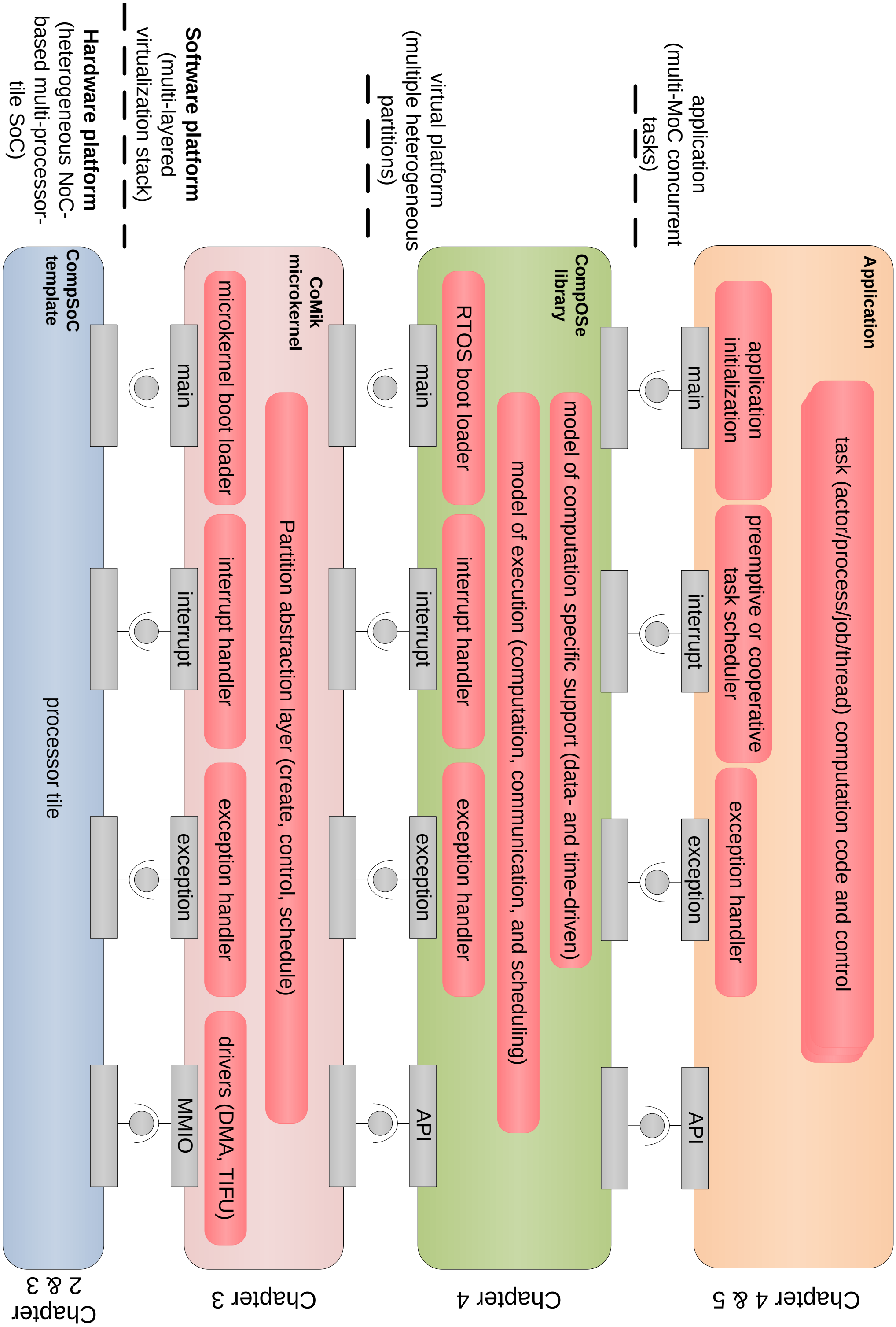

application
(multi-MoC concurrent tasks)
Application
task (actor/process/job/thread) computation code and control
application initialization
preemptive or cooperative task scheduler
exception handler
main
interrupt
exception
API
Chapter 4 & 5
virtual platform
(multiple heterogeneous partitions)
CompOSe library
model of computation specific support (data- and time-driven)
model of execution (computation, communication, and scheduling)
RTOS boot loader
interrupt handler
exception handler
main
interrupt
exception
API
Chapter 4
Software platform
(multi-layered virtualization stack)
CoMik microkernel
Partition abstraction layer (create, control, schedule)
microkernel boot loader
interrupt handler
exception handler
drivers (DMA, TIFU)
main
interrupt
exception
MMIO
Chapter 3
Hardware platform
(heterogeneous NoC-based multi-processor-tile SoC)
CompSoC template
processor tile
Chapter 2 & 3

Figure 1.2: An overview of the composable virtual platforms for mixed-criticality embedded systems.

To create a composable temporal partition of the processor tile, the arbitration scheme has to provide a guarantee on when the service is exactly available to a partition in order to prevent the interference of other partitions. Every application executes in a dedicated partition. Time Division Multiplexing (TDM) is used as one of the arbitration schemes that can provide such a guarantee. Using this technique a set of time slots are created as fixed resource utilization time quanta. A temporal partition is then a set of these time slots allocated to an application. Applying the composable partitioning technique to all the resources involved in executing an application in a processor tile, a partition is formed. A VP is then a set of these partitions allocated to one application from resource in the platform.

On top of the hardware platform, a privileged software layer is designed to provide the essential services of partitioning for the purpose of virtualization. Our solution realizes this layer in the form of a microkernel, namely CoMik. CoMik virtualizes the underlying platform by executing the instruction routines in the form of three software hooks, i.e., main function, interrupt handler and exception handler, as illustrated in Figure 1.2. Using these, CoMik creates, controls, and schedules processor tile partitions. Moreover, it executes an application in its partition by virtualizing the software hooks and offering an API so that the application could use its allocated resources.

Created isolated partitions by the CoMik microkernel, an application can execute directly on its virtual platform. For this, the model of computation has to use the primitive execution operations offered by CoMik via the partitions. In our solution, a model of execution is however proposed to fill the gap of execution abstraction between the models of computation's semantic and the platform's primitive operations. The model of execution categorizes the execution operations as: (i) computation, (ii) communication, and (iii) scheduling operations.

The model of execution targets Kahn Process Network (KPN) and dataflow (specifically, Cyclo-Static DataFlow (CSDF)) as the two data-driven models of computation, and time-driven models of computation. Supporting these, a wide range of time-criticality application domains is covered. Data-driven models of computation typically require to schedule their tasks cooperatively, while the time-driven model needs a preemptive type of scheduler. The model of execution therefore proposes a unified manner to implement both data-driven and time-driven models of computation, and for this, it identifies the common operation primitives of these models.

The model of execution is implemented in form of a lightweight Operating System (OS) library, namely CompOSe. The CompOSe library is instantiated in every partition of the VP created by CoMik for an application. As illustrated in Figure 1.2, CompOSe is designed in a number of software units for: (i) providing the *main*, *interrupt handler*, and *exception* handler software hooks required by CoMik from a partition, (ii) implementing the execution primitives of the model of execution using the API provided by CoMik, and (iii) giving model of computation specific support to the application by providing Software Containers (SCs). CompOSe is implemented in such a way that it does not introduce any unpredictability in executing an application on top of the CoMik microkernel and therefore it complies with predictability and composability of the system.

Finally, at the application layer, multiple applications can be expressed in different models of computation each of which execute in a partition of a dedicated VP. The applications may use its own arbitrary type of task schedules, and even handle their own interrupts and exception in fully isolation of the other concurrent ones.

1.3 Contributions

This dissertation makes six contributions to develop the solution of providing Composable VPs for mixed-criticality systems, as follows.

1. The temporal partitioning technique is applied to the CompSOC embedded template for the purpose of creating composable VPs. For this, a Timer-centric Interrupt and Frequency Unit (TIFU) is developed. This hardware unit enables composable temporal partitioning of the processor, offers isolated virtual time management of the applications execute in a partition, and provides interrupt virtualization support[1].

2. The CoMik microkernel is designed and implemented as virtualizing software layer that creates strongly composable temporal partitions for each application, and abstracts the execution operations of the underlying platform by providing a set of higher level of abstraction API to the applications execute in their partitions. CoMik implements a cycle-level composability between the partition without needing to modify the processor architecture[1].

3. The unified model of execution is proposed to fill the gap of execution abstraction between the models of computation's semantics and the primitive execution operations supported by the VPs. It realizes the models of computation by using their common execution operations and representing them with a sequence of regular expressions.

4. The CompOSe library is designed as a lightweight OS to implement the model of execution. It is designed to run as an untrusted code in a partition and accesses the underlying platform resources in controlled and restricted mode. By the set of API and SC that it offers to applications, multiple models of computation with different time-criticality can be implemented on the platform.

5. The timing properties of the proposed composable platform are formalized by introducing a response time analysis of time-driven applications to check for their schedulability on the CompSOC platform. The formalization enables application developers following two design options: (i) (re-)design (legacy) applications to be schedulable on the CompSOC platform, and (ii) adjust the partition parameters, i.e., slot allocations and slot sizes, so that the available (legacy) applications would be schedulable.

6. The predictability, the composability, and the support of multiple models of computation are demonstrated empirically by using two major experiment setups. One, a Matlab simulation of the virtual platform is prepared to investigate the

[1] This contribution is a joint research work done in collaboration with Andrew Nelson who also covers this in his dissertation [75].

temporal behavior of CoMik in partitioning the processor tiles. Two, an FPGA prototype of the platform is used to study the composability property and support of multiple models of computation. For this, a number of use-cases consisting of real and synthetic applications execute on an MPSoC instance of the CompSOC platform.

1.4 Organization

In the rest of this dissertation, the solution is organized and presented according to Figure 1.2. Immediately after this introduction chapter, Chapter 2 gives an overview of the background information, starting with the definition of the predictability and the composability properties for embedded systems. Afterward, it introduces how these properties is implemented in the existing template of CompSOC architecture. The chapter provides also a detailed description of data- and time-driven models of computation.

Chapter 3 first motivates the virtualization technique as the solution for composability. Then it proposes partitioning techniques that can be used for virtualization. Finally, it described the details of CoMik software architecture.

Chapter 4 presents the model of execution and its implementation in the form of the CompOSe library. It details the complete structure of CompOSe software units and how an application can be implemented using this library.

Chapter 5 first proposes a response time analysis of time-driven applications to check for their schedulability on the CompSOC platform. Second, it demonstrates the predictability, the composability properties of our system, and the support of multiple models of computation by Matlab simulation and Field Programmable Gate Array (FPGA) emulation of the platform. It presents the empirical results of some real and synthetic application use-cases.

Chapter 6 concludes this dissertation with respect to the contributions introduced in the introduction chapter. It proposes several opportunities for future research work.

CHAPTER 2

Background

THE main objective of an embedded system is to carry out a set of algorithmic tasks realizing a number of applications. For this purpose, system developers have to follow three main steps. These steps are presented in Figure 2.1 together with their corresponding levels in an embedded system stack.

At the *design* step, the applications' tasks are expressed in high-level algorithms. At the *implementation* step, every application is then implemented using a model of computation. At the *execution* step the application is executed on the hardware/software platform. The *execution* step refers to two levels in the system stack: a model of execution and the execution platform. A model of execution fills the gap of execution abstraction between the model of computation's semantics and the primitive execution operations supported by the platform.

Nowadays, embedded applications have a time-criticality level, i.e., real-time or non real-time requirements. A mixed critical system therefore has applications with more than one criticality level. In case of real-time applications, the system has to ensure that the timing requirements of the applications are met. For this purpose, the system has to be designed in such a way that the timing characteristics of the system are so well defined that the temporal behavior of the applications can be verified against their requirements before actually executing on the platform. Resource sharing between applications with mixed time-criticality concurrently executing on a single platform causes interference between the applications. The interference may cause unpredictable (temporal) behavior of the applications. In order to prevent this, the applications have to execute temporally isolated.

Predictability and *composability* are two system properties that are proposed to deal with these challenges [85]. As depicted in Figure 2.1, predictability applies to the implementation and the execution steps. It therefore covers the model of computation, the model of execution and the platform layers of the system stack. Composability instead involves the layers that are shared between the applications, i.e., the model of execution and the platform layers.

In this chapter, we first define the predictability and the composability properties for an embedded system. Following that, starting from the bottom of the stack, we elaborate on the state-of-the-art approaches that are used to realize the predictability and

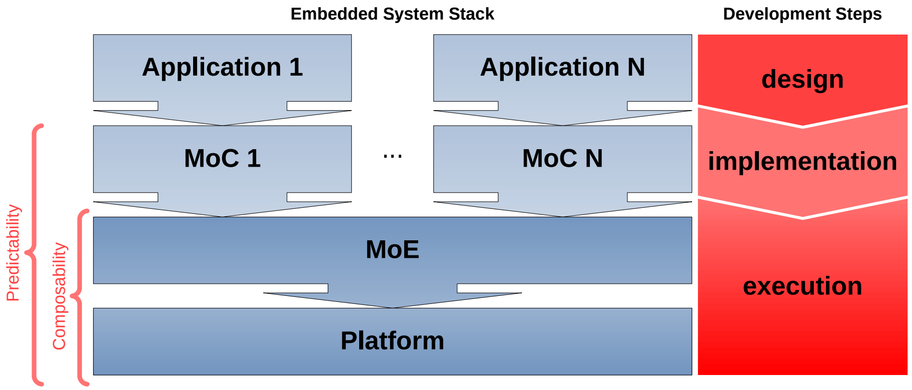


Figure 2.1: *Predictability and composability properties of the embedded system stack layers.*

the composability properties at each layer. The main contribution of this dissertation is in the model of execution layer, hence, it will be discussed separately in the later chapters.

2.1 Predictability

Predictability is a property of a system and is defined as a level of how well the timing characteristics of the system are defined and implemented so that the system temporal behavior is known before it actually starts running the applications [10]. In other words, a system is predictable if it is possible to derive a temporal-behavior model of the system so that the timing requirements of the running applications can be verified at design time [35].

A predictable temporal model can be either very detailed such that it corresponds to the actual case of the system, or less detailed such that provides a bounded worst-case behavioral model of the system [95]. In the detailed model, the temporal behavior of every resource of the system at each moment in time is known and therefore an exact execution trace of an application's temporal behavior can be derived from this model. Whereas in the bounded model, the worst-case temporal behavior of the system's resources can be extracted, and therefore, the worst-case temporal behavior, i.e., minimum throughput and maximum latency, of applications is known. In order to guarantee real-time requirements of applications executing on dedicated resources of a system, it is necessary and enough to come up with the worst-case behavioral model for all the components involved in execution of applications.

Predictability is a bottom-up property, meaning that it is not possible to have a predictable system developed on top of an unpredictable resource. Therefore, in the embedded system stack illustrated in Figure 2.1, every resource of the execution platform and the model of execution has to be predictable so that the temporal behavior of the applications implemented in the models of computation is predictable, as well.

Later, this chapter introduces the architecture of the predictable execution platform and the models of computation that are used to implement the applications with mixed temporal criticality.

2.2 Composability

To reduce the cost when a number of applications execute concurrently on a single platform, the resources are shared between the applications. Resource sharing causes temporal interference between the applications. In a mixed-criticality system, such interference might be either predictable or unpredictable. The predictable interference is typically between the real-time applications with the bounded temporal execution behaviors; whereas the unpredictable interferences result from the non-real-time applications. In case of predictable interference, as the number of concurrent execution combinations of the applications may grow exponentially, the verification and integration would become dramatically difficult [53]. In the case of unpredictable interference, it is not possible at all to verify the timing requirements of the real-time applications and therefore to integrate them with the non-real-time ones.

Composability is the property that the temporal (and functional) behavior of an application is isolated and independent from of other concurrent applications. A system is either *weakly* or *strongly* composable. Traditionally, weak composability is denoted as a relaxed isolation between applications, where with the help of predictability a system could make the worst-case temporal behavior of an application unaffected by the other applications [55]. This enables a compositional computation of worst-case bounds of the real-time applications. However, in the systems with mixed criticality applications the weak composability does not help to reduce the complexity of system verification and integration as whenever one of the applications changes or a new application has to be integrated with the existing ones, the overall process of verifying all the applications' timing requirements has to be repeated.

Recently, a stricter definition of composability, which is orthogonal to predictability, is proposed as *strong* composability [4, 26]. In this definition, the isolation between the applications is in such a way that the *actual-case temporal behavior* of each application is unaffected, i.e., the exact starting, finishing, and actual and worst-case timing properties of an application are *cycle-accurately* independent of the other applications. This enables independent design, verification, and integration of mixed-criticality applications. From now on, in this thesis, where we denote composability we mean such a strict definition.

Since composability is defined as an inter-application property, it only applies on the layers of the system stack that deal with shared resources between the applications. These are the model of execution and the platform layers in Figure 2.1. The layers have to implement the resources or manages the access of the applications to resources such that the temporal isolation between the applications are guaranteed at cycle-level. The technology to implement composability differs per resource type. Later, this chapter introduces how the composability is implemented in the existing architecture of the execution platform.

2.3 Hardware Platform Architecture

Shrinking the feature size of the transistors has led to low-cost integration of more and more number of processing resources on a single chip and consequently allows designing complex chip multiprocessor architectures [44]. However, since the requirements of applications grow ever faster, these architectures are still resource constrained. For cost reasons, the applications executing on these platforms have to share some resources, such as processors, interconnect, memory blocks, etc.

In order to guarantee predictable and composable execution of applications with mixed time-criticality, each resource has to be specifically designed and implemented for this purpose. CompSOC is a system-on-chip template developed following composability and predictability paradigm [29]. In the rest of this section we are going to describe the CompSOC hardware architecture as the platform layer of the system stack depicted in Figure 2.1. All the other layers are developed on top of the CompSOC architecture.

2.3.1 CompSOC: A Composable & Predictable System-on-Chip Platform

The CompSOC template implements a tile-based multi-processor architecture which consists of a number of processor and memory tiles communicating via an on-chip interconnect, as depicted in Figure 2.2. Multiple applications may run on each processor tile and the communicating tasks of one application may be mapped onto multiple processor tiles. Thus, the processor tiles, the interconnect, and the memory tiles are possibly shared between and within the applications. *All these resources have to be designed and arbitrated in such a way that they realize predictable and composable execution of applications.* In the rest of this section, we are doing to describe the detailed architecture properties of these resources.

Processor Tile

A processor tile consists of a processing core, a data and an instruction memory, a number of communication memory blocks, a Programmable Interrupt Timer (PIT), a Clock Control Module (CCM), and a number of Remote Direct Memory Access (RDMA) modules each of which is equipped with a dedicated memory block.

The processing core is a MicroBlaze [101] embedded soft core which is highly configurable for some specific set of features that are necessary for the purpose of predictability and composability. The Microblaze is a 32-bit Reduced Instruction Set Computer (RISC) architecture, optimized to be implemented on Xilinx Field Programmable Gate Arrays (FPGAs) [100]. The processor can be configured for either three or five stage instruction pipelining.

In the CompSOC platform, the instruction cache is disabled to remove the interdependency between the applications due to cache misses/hits. The instruction code of the applications are therefore stored locally in an instruction memory (*imem* in Figure 2.2), which is single-cycle accessible via the Instruction Local Memory Bus (ILMB). Ideally private data of an executing application on a tile should also reside in the local data memory (*dmem* in Figure 2.2) of the tile to minimize the data access latency. However, practically an embedded application has to transfer data to/from other tile's memory locations (when part of the application is mapped and executed on another tile), to/from off-tile memory blocks such as Double Data Rate (DDR) Synchronous Dynamic Random-Access

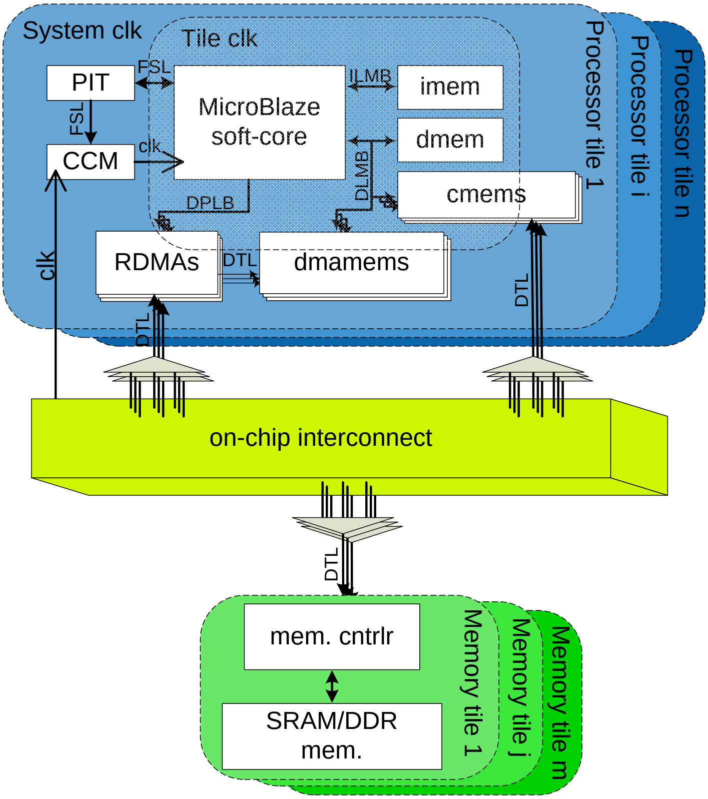


***Figure 2.2:** The existing hardware architecture of the CompSOC platform.*

Memory (SDRAM), or Memory-Mapped Input/Outputs (MMIOs). For this purpose, the application initiates data read/write transactions over the interconnect.

When the processor directly performs such a data transfer over the interconnect, it blocks until the transaction finishes. As the size of data being transferred may be variable (or even infinite, for example in case of continuously streaming in/out data), the blocking time is not known a priori. It is even worse when the processor executes a misbehaving application: the processor may be blocked indefinitely. This compromises the composability of the system when multiple applications execute concurrently on the processor.

For such off-tile communication, RDMA modules, each of which is associated with a memory block, denoted as *dmamem* in Figure 2.2, are used. Every RDMA is assigned to one application and can be programmed by the processor via the Data Processor Local Bus (DPLB) to transfer data (with the granularity of 1 to 64 words) from/to a location in its associated *dmamem* to/from another memory-mapped location through the interconnect, e.g., remote memory of other tiles. After being programmed, each RDMA performs off-tile communication independently of the application executing on the processor, and its status may be read by the application to check if the transaction is done.

Furthermore, other tiles in the system can use MMIO to transfer data to/from the tile through communication memory blocks, denoted as *cmems* in Figure 2.2. These dual port memories, from one side, are locally accessed by the processor via the Data Local Memory Bus (DLMB) as scratchpad memories, and from the other side, they interface with the interconnect via a Device Transaction Language (DTL) bus [80].

Traditionally, to protect against unwanted memory accesses by a running application, a Memory Management Unit (MMU) accompanies the processor. For the CompSOC platform, a predictable and composable MMU is proposed in [64]. Due to a large cost of this scheme, the module is not regularly instantiated in the platform. For stack protection, however, the Microblaze has an internal feature which is always enabled in the CompSOC platform. In the next chapter, we present how this feature is used by the software architecture.

The PIT, which is programmable from the processor via a Fast Serial Link (FSL), is used for two purposes: (1) to keep track of the time, and (2) to generate an interrupt at a given moment in time. The first one is necessary for performance monitoring of real-time applications. The latter is essential for implementing timed-event based functionality in the system. Keeping such basic functionality of the PIT, in this thesis, we later replace this component with a more advanced module, namely Timer-centric Interrupt and Frequency Unit (TIFU).

Every processor tile can run on a different clock frequency, for example, for the purpose of power management [76]. In an FPGA prototype of the CompSOC platform, this is emulated by the CCM which can divide the system clock frequency and provide the tile clock. To perform such clock division, the CCM has to be tightly coupled with the PIT. This technology leads to having two clock domains in every tile, as illustrated in Figure 2.2, with the tile clock and the system clock.

Memory Tile

The memory tile consists of a front-end memory controller and a back-end memory technology which can be a SRAM and/or DDR. The memory controller arbitrates between the number of requester connections that want to access the back-end memory in such a way that the accesses are predictable and composable [3]. The details are out of scope of this dissertation, and for further information we refer to the literature [2].

On-Chip Interconnect

The interconnect consists of traditional bus technology and the Æthereal Network-On-Chip (NOC) architecture [22, 27, 31, 89]. A connection over this interconnect is recognized as a request channel from a master port to a slave port, and a response channel from a slave port to a master port. A master port may have a connection to multiple slaves, and multiple masters may also have connections to a single slave. In these cases, a master-bus at the master side and a slave-bus at the slave side are used to de-multiplex and multiplex the connections, respectively.

The basic building blocks of the NOC are *Network Interfaces (NIs)* and *routers*. The topology of Æthereal is arbitrary, where more than one NI may connect to a router. The physical communication NI-router and router-router links are shared between multiple connections. In order to provide predictable and composable communication for the connections, a circuit switching routing algorithm is implemented with a Time Division Multiplexing (TDM) scheme at NI sides. A design-time tool flow [28] calculates the size of the TDM slot table and allocates the slots to each connection according to the given throughput and latency requirements of each connection. In this way, a bound on latency and throughput of the connections over the interconnect is guaranteed.

Summary

The underlying execution architecture of the platform is a tile-based System-on-Chip (SoC) with distributed shared memory. Every component of the architecture and the overall platform is composable and predictable. The architecture is designed following the Asymmetric Multiprocessing (AMP) scheme [62]. It means that the processor tiles can be architecturally different. From hardware architecture perspective, at design time, each processor tile can be configured to include different number of RDMA modules, different number of communication memory blocks, and different instruction, data, communication and RDMA memory sizes [23]. Besides this, if required by an application, any processor tiles can be equipped with dedicated hardware peripherals such as I/O device controllers, e.g, TFT or Ethernet. From the software architecture perspective, each processor executes its own instruction code resident in its local instruction memory (*imem*), and can communicate via MMIO data transactions to distributed shared memories.

The scope of our work from now on is the composable and predictable execution technology on a single processor tile. The next section goes one layer higher in the system stack of Figure 2.1 to describe the preliminaries of executing the applications on a processor tile.

2.4 Application Execution

Traditionally, a software stack running directly on top of a hardware infrastructure is responsible for managing the execution of the applications to comply with the application's execution requirements. Such software stack typically is an Real-Time Operating System (RTOS) including drivers of the underlying hardware components. One of the main responsibilities of an RTOS is to schedule different software components on the platform. In this section we present the preliminaries of scheduling operations on an embedded system. Based on this discussion, in the following chapters, we present our software architecture for CompSOC platform. On top this architecture we will then propose a model of execution.

2.4.1 Scheduling

Scheduling is to allocate time for use of a resource to a set of requesters. In case multiple resources are available, a scheduler has to first assign a resource to the requester and then allocate the time.

For us, the finest grain granularity of a requester is a task, i.e., a task is a scheduling unit for a processor. An application is then a set of communicating tasks. When multiple applications are to execute concurrently on a multi-processor platform, a set of the applications' tasks have to be scheduled on the processors.

Global Vs. Partitioned Scheduling

Traditionally, the multi-processor scheduling can be done by following either a global or a partitioned scheme [15]. In the global scheme, a central scheduler (as part of a global Operating System (OS)) first assigns the tasks to the processors and then schedules them to execute on the processors. In a partitioned scheme, the tasks are statically assigned to the processors and a separate scheduler per processor allocates the processing time to the tasks. The partitioned scheme is scalable comparing to the global scheme and fits the best to the AMP scheme because the separate instance of the scheduler on the processors can potentially use different scheduling policies. Thus, the software stack of the CompSOC platform has to implement the partitioned multi-processor scheduling scheme.

Hierarchical scheduling

The statically partitioned set of tasks that are allocated to a processor tile may belong to different applications. In such a case, conventionally, the scheduling is done at two hierarchical levels [66]. At the first level, an inter-application scheduler selects an application for which at the second level, an intra-application scheduler assigns the processor time to one of its task.

Considering the embedded stack presented in Figure 2.1, the composability is at the application-level. Therefore, when following hierarchical scheduling scheme, only the inter-application scheduler has to comply with composability property. This allows the applications to use different intra-application schedulers for their own purposes. In the following chapters, we propose a software architecture for the platform and we discuss the implementation details of the hierarchical scheduling scheme.

Scheduling Classifications

Generally, based on the type of the triggers that invoke the schedulers, two scheduling schemes can be distinguished: (1) time-triggered, i.e., when the scheduler is invoked by a timed event, e.g., timer interrupt; (2) event-triggered, i.e., when the scheduler is invoked by any other event than a timed event, for example, an I/O interrupt or data availability.

Furthermore, scheduling can be either preemptive or cooperative with respect to how the scheduler deals with the executing task. If the invocation of the scheduler preempts the executing task, the scheduling is preemptive; whereas, in case of cooperative scheduling, the executing task is allowed to finish (or signal that it can yield the processor) before the scheduler is invoked.

The invocation schemes of schedulers are orthogonal to their cooperativeness. This means that event- or time-triggered schedulers can be preemptive or cooperative. Preemptive schedulers are invoked immediately when the triggers are raised, while in case of cooperative ones, a time delay may exist between the time that a trigger is raised and the time that the scheduler is invoked.

From an application perspective, these schemes affect its task's execution timing. We will discuss the effect of each scheduling scheme as part of our model of execution proposed in the future chapters. For this purpose, in the rest of this section, we introduce the applications' task temporal model that we consider in this thesis.

2.4.2 Task Temporal Model

In an execution platform, a task is a piece of sequential code that implements a part of an application. It is then translated into a sequence of processor instructions that executes as the body of the task, implementing its functionality. This sequence of instructions are known as the task's *workload*, denoted as C, and measured in number of processor clock cycles needed execute the code entirely. In case of real-time applications, every task's (worst case) workload has to be known before the task actually executes on the processor. Whereas for non-real-time applications this is often not the case.

Assuming that a task with workload of C cycles executes on a processor, we define the execution temporal model of the task as illustrated in Figure 2.3. In what follows, the characteristics of the temporal model for a task (τ_i) are described.

release time (also known as arrival time) (r_i): the time moment at which the task is ready to execute.

start time (s_i): the time at which the task receives the control of the processor and starts to execute.

finish time (f_i): the time at which the task finishes.

due time (also known as deadline) (d_i): the time before which the workload of the task must be finished. In real-time applications, the deadline is a key requirement that imposes constraints on the execution of the tasks. Tasks of non-real-time applications do not have deadlines.

preempt time ($\boldsymbol{pr}_{ij}$): the jth time at which the executing task is preempted and the task loses the control of the processor. Note that a task may never get preempted during its execution.

resume time ($\boldsymbol{rs}_{ij}$): the time at which the task preempted at the time pr_{ij} receives back the control of the processor to resume its execution.

(actual) execution time ($\boldsymbol{et}_i$): the sum of all the time durations in which the tasks has been executing and it is equivalent time to the task's workload. Mathematically, assuming that the task is preempted n times, and $rs_{i0} = s_i$ and $pr_{in} = f_i$, then, $et_i = \sum_1^n (pr_{ij} - rs_{i(j-1)})$.

logical execution time ($\boldsymbol{let}_i$): the duration between the finish time and start time of the task, i.e., $let_i = f_i - s_i$. It is called logical time since the task has been actually executing only in some parts of this time.

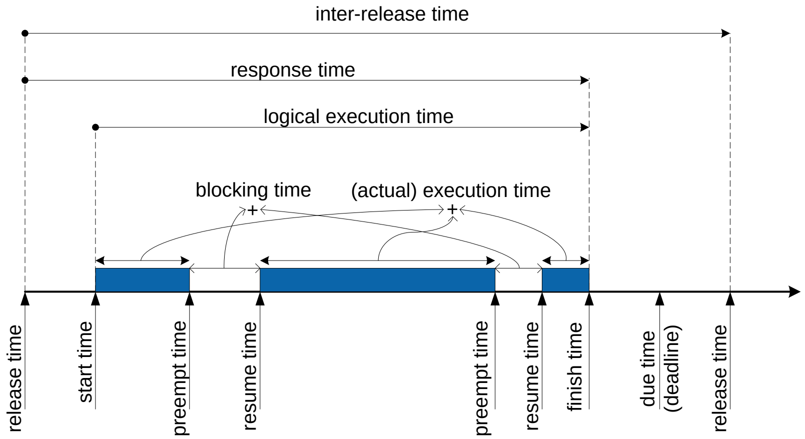


Figure 2.3: A Task execution temporal model.

blocking time (bl_i): the sum of the all time durations in which the task has been preempted. Mathematically, $bl_i = let_i - et_i$.

response time (rsp_i): the duration between the finish time and release time of the task, i.e., $rsp_i = f_i - r_i$.

inter-release time t_i: the duration between the two consecutive release times of the same task. Depending on the task's model of computation, its inter-release time could be periodic, aperiodic, or non-periodic.

Such a temporal model of a task is general and it has different interpretations in different applications' models of computation. To show this, in the following section, we introduce models of computation that we use in this thesis to implement the applications.

2.5 Model of Computation

At the top of the embedded system stack in Figure 2.1, an application is a set of algorithmic computational operations that realize a functionality. To execute each application on the platform, a model of computation implements the application using a set of formal semantics defined to express the computational operations [43].

As embedded applications belong to various domains such as consumer electronics, automotive electronics, avionic, industrial control, medical electronics, etc., the applications realize different functionality with different intrinsic behaviors. For example, multimedia applications have different functional behavior comparing with an engine control application. One example of multimedia applications is a video codec that typically receives an streaming input of data on which it applies some computation and outputs an stream of data for illustration. Instead, the engine control application, as an example of automotive applications, periodically reads some sensors, applies some computation, and produces a signal to control an actuator at specific moment in time. Thus, different models of computation are required to implement different applications.

Traditionally, the applications are first implemented in a sequential model of computation using an imperative programming language such as C. Due to recent advances in execution platforms that enable concurrent execution of multiple tasks, concurrent (parallel) models of computation are used to implement an application by parallelizing its functionality into a number of individual tasks.

Depending on the intrinsic behavior of an application, its tasks may synchronize naturally either on the basis of data availability or on specific times. In other words, the tasks may be either data- or time-interdependent. Accordingly, two different models of computation, namely data-driven and time-driven, are used to implement such interdependencies between the tasks.

In this section, we first introduce these two types of models of computation. Second, the predictability of the models is discussed. Finally, we conclude this section by presenting how each model of computation realizes the tasks based on the model presented in the previous section.

2.5.1 Data-driven Model of Computation

In the literature, various parallel models of computation are proposed for implementing data-driven applications [92], also known as streaming applications. Each of these models has its own properties that makes it suitable for various application domains [57]. One of the two important properties of these models is 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. The second property is analyzability for predictability, i.e., how amenable the model is for timing analysis so that the minimum throughput and maximum latency could be estimated. For example, in a real-time application, once started, tasks may execute without any blocking, and therefore, a worst-case bound for their logical 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 data-driven models of computation is presented in [93]. Here, Figure 2.4 presents this comparison among the sequential programming models, the Kahn Process Network (KPN) model and three variants of dataflow model, namely Static Dataflow (SDF), Cyclo-Static DataFlow (CSDF), and Variable Rate Dataflow (VRD). In general, sequential models of computation have no restrictions in using the primitives of imperative programming languages such as C, and therefore they can be highly expressive in modeling different behavior of applications, except for parallelism, but the analyzability of the models may be lost.

Adhering to coding restrictions, sometimes a sequential model of computation can be automatically transformed into parallel models of computation [50, 97], such as *Kahn Process Network* (KPN) [48], and *dataflow* [56], and can be used in Firm Real-time (FRT), Soft Real-time (SRT), and Non Real-time (NRT) application domains.

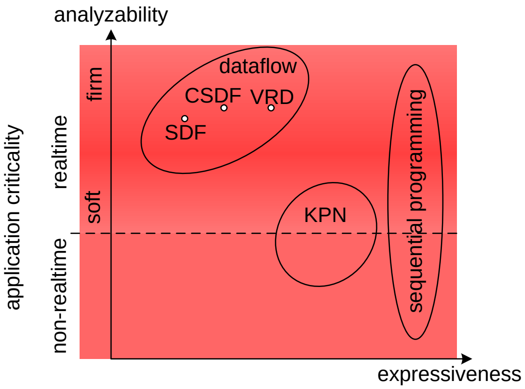


Figure 2.4: *Analyzability versus expressiveness for common data-driven models of computation.*

The KPN and dataflow models are networks of concurrent *nodes*, referred to as *processes* in KPN and as *actors* in dataflow terminology [18]. A node is a functional mapping from inputs to outputs. 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 Figure 2.5.

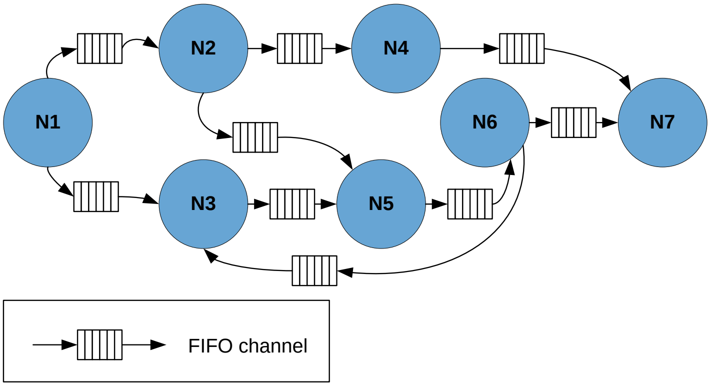


Figure 2.5: *Node graph of a data-driven model of computation.*

In the following subsections, we describe the characteristics of the KPN and the dataflow models in details.

Kahn Process Network

A KPN process body, presented in Listing 2.1, 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 [90]. A process blocks on a read or write when the FIFOs does not have enough data or space, respectively. With limited size FIFOs, the presence or absence of deadlock

```
/* Start of process body */
initialization();

For (i=0; i<N; i++) {
  x = read(<f4,1>); /* Read 1 data token from FIFO 4 */

  P5_compute1(x);

  if (x == 0) {
    y = read(<f5,1>); /* Read 1 data token from FIFO 5 */
  } else {
    y = P5_compute2(x);
  }
  z = P5_compute3(y);

  write(<f6,1>, [z]); /* Write 1 data token to FIFO 6 */
}

write(<f6,1>, [a_final_token]); /* Write 1 data token to FIFO 6 */

finalization();
/* End of process body */
```

Listing 2.1: *An example pseudo-code of a process in a KPN model of computation.*

can be computed at design time for any KPN graph [16]. A KPN process is activated once and the process itself has to implement any iterative execution, as presented in Listing 2.1 with the main loop inside the body.

Dataflow

A dataflow actor body is a sequence of `consume`, `compute`, and `produce` operations, in this strict order, as presented in Listing 2.2. 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. For example, in Listing 2.2, the firing rules of the task check for different number of tokens on FIFO 5 and 6, in its two cycles. Once the firing rule is satisfied, an actor executes its entire body without blocking on input or output. The actors is activated whenever their firing rules are satisfied. An actor executes for an infinite number of activations. In dataflow, each activation of an actor corresponds to one iteration over its body.

Different variants of dataflow models exist, e.g., SDF, CSDF [79], Variable Rate Dataflow (VRD) [99]. SDF and CSDF are analyzable, i.e., the worst-case timing behavior of their graph can be computed, when their sequential actors have bounded execution time. This is due to the fact that the execution time of a dataflow actor does not depend on its communication with the other actors and therefore after the actor firing rules are satisfied, the input an output tokens are ready for the entire execution of it computation without any blocking. Thus, given the worst-case execution time of all actors, the worst-case latency and throughput of the graph can be calculated. Given a predictable Multi-Processor System-on-Chip (MPSoC) platform, the communication time between the actors can be also bounded. Therefore, existing formalisms can analyze an application and derive an end-to-end latency, throughput, and buffer sizes for it [12].

```
/* Cyclo static firing rules */
switch cycle
  case 0:
    wait_until ( data_token_available_on_fifo(<f4, 1>, <f5, 1>)
                 AND
                 space_token_available_on_fifo( <f6, 2>) );
  case 1:
    wait_until ( data_token_available_on_fifo(<f4, 1>, <f5, 0>)
                 AND
                 space_token_available_on_fifo( <f6, 1>) );
End

/* Start of actor body */
  x = consume(<f4,1>);  /* Consume 1 token from FIFO 4 */

  IF (cycle == 0) y = consume(<f5,1>); /* Consume 1 token from FIFO 5 */

  w = P5_compute1(x);

  IF (cycle == 0) z = P5_compute2(y);

  IF (cycle == 0) {
    produce(<f6,2>, [w,z]); /* Produce 2 tokens to FIFO 6 */
  } ELSE {
    produce(<f6,1>, [w]);   /* Produce 1 token to FIFO 6 */
  }

/* End of actor body */

cycle = (++cycle) MOD 2; /* Increment cycle for the next activation */
```

Listing 2.2: *An example pseudo-code of an actor in a CSDF model of computation.*

Our target platform can execute all three mentioned variants of the dataflow. However, in this dissertation, we only focus on CSDF. In CSDF every actor has a set of cyclically changing static firing rules. As illustrated in Listing 2.2, depending on the cycle of the firing-rules set that an activation is in, a static firing rule checks different FIFOs against the expected number of available tokens.

Analyzability Vs. Expressiveness

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. 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 timing analysis required for FRT applications for exactly those reasons. 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, which is acceptable for

SRT applications [16]. Thus KPN can only fit NRT and SRT applications.

The execution of both KPN and dataflow models on an MPSoC platform enables our model of execution to support several application domains.

2.5.2 Time-Driven Model of Computation

Traditionally, real-time applications are implemented using time-driven models of computation. In this model, an application's functionality is split into a number of (concurrent) *jobs*. Each job is ready to execute at an specific moment in time, when it assumes the data and the space that it may require for its execution are available.

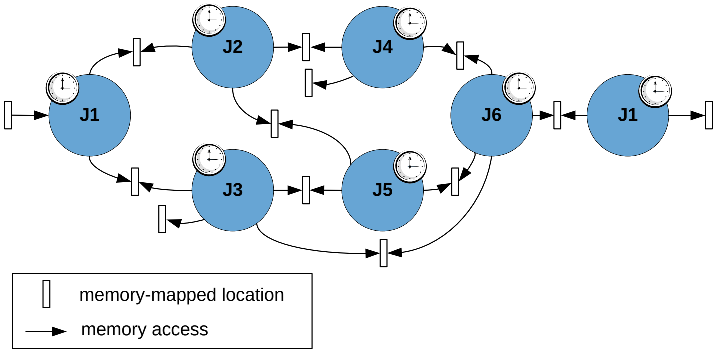


Figure 2.6: *Job graph of a time-driven model of computation.*

The jobs may communicate with each other through memory-mapped data locations, as illustrated in the job graph of Figure 2.6. In this model, no job blocks on data or space that it needs for its execution. Thus, the applications implemented with this model is predictable, if the (worse-case) execution time of each job is bounded. A pseudo code example of a time-triggered job is presented in Listing 2.3.

If there is more than one job ready to execute at a time, the one that has given higher priority starts executing and it may preempt a running lower priority job.

```
/* Wait until a given time to fire */
wait_until (time);

/* Start of job body */
x = *memory_location_ptr_1;

y = J7_compute(x);

*memry_location_ptr_2 = y;

/* End of job body */
```

Listing 2.3: *An example pseudo-code of a process in a time-triggered model of computation.*

2.5.3 Summary

In this section we presented two common models of computation, i.e., data- and time-driven, that are used to implement real-time applications. In these models, each application is parallelized into a number of concurrent process, actor, or jobs.

To execute the applications on the platform, each of the processes, the actors, and the jobs, have to be realized by the task model introduced in Section 2.4.2. However, some of the general task characteristics refer to different intrinsic behavior of tasks in each model of computation. Here, we explain the most important ones.

In the KPN model, every process corresponds to a task. As there is no specific condition for a KPN process to start executing, it is released once the system starts, and it may or may not finish its execution in the system life-time. No deadline and inter-release time is therefore defined in this case. The performance of an application depends on its actors' execution times.

In the CSDF model, every actor corresponds to a task. An actor is released when its firing rules are satisfied. The minimum inter-release time of an actor is therefore equal to its minimum response time, i.e., an actor is released immediately after it finishes one iteration if the firing rules are satisfied. Similar to the KPN process, no deadline is defined for CSDF actors, and the performance of an application depend on its actors' execution times.

In the time-driven model, every job corresponds to a task. The release time, deadline, and inter-release time of the jobs are given at design time. If the the inter-release time is always fixed the task is released periodically, whereas, in the case of variable inter-release time the task is aperiodic.

Considering all these characteristics of the models of computation tasks and based on the scheduling classification presented earlier in this chapter, we can summarize common scheduling schemes in each model of computation in Table 2.1.

In the KPN and the CSDF models production of data tokens in FIFOs are the events that can trigger the schedulers cooperatively. In KPN, process may also block at any moment in time on a FIFO read or write. In this case, the process either cooperatively yield the control of the processor to the scheduler or it waits until an interrupt preempts it. The preemption is necessary to avoid deadlock. In CSDF, yielding or preemption is not necessary since actors do not block on acquiring tokens at all. In the time-triggered model, unless execution of jobs are exclusive in time, preemptive task scheduler is required.

In the later chapters, where we propose our model of execution, we explain in details how the KPN processes, CSDF actors, and time-driven jobs are actually implemented and executed in our execution platform, and we reason further how a scheduling scheme suits a model of computation the best with respect to Table 2.1.

Table 2.1: *Task scheduling overview of the models of computation.*

Task	Scheduling Invocation Scheme	Scheduling Class
KPN process	Event-Triggered	cooperative or preemptive
CSDF actor	Event-triggered	usually cooperative, sometimes preemptive
Time-driven job	Time-Triggered	(almost) always preemptive

CHAPTER 3

Composable Virtualization

Virtualization is the technology to provide an illusion of execution resources to applications so that the applications behave like running directly on the bare resources [1, 33]. Such an illusion is provided by creating a virtual instance of the actual platform, known as Virtual Machine (VM) or Virtual Platform (VP), by a Virtual Machine Monitor (VMM) or a hypervisor [41].

The objective of virtualization is different from one application domain to the next [32, 51]. The most important ones can be enumerated as: (1) isolating applications execution, (2) load balancing, (3) power management, (4) security, (5) supporting legacy applications by running different OSs. In this work, we use virtualization technology to perform predictable, composable verification, integration, and concurrent execution of mixed-criticality applications on the CompSOC platform. Moreover, this also enables us to run different OSs concurrently on the same platform.

Depending on whether the VP is created directly or indirectly on the hardware platform, the Virtualization technology is classified into two types. In virtualization of type-I, the VP is created directly on the hardware platform, whereas in type-II, an OS typically hosts the VP. Due to resource constraint in embedded systems, the type-I VP is typically created by a microkernel. A microkernel is a minimal super-privileged software layer that provides only essential services of partitioning for the purpose of virtualization [34].

In the context of the CompSOC platform, we propose a microkernel, namely CoMik, to achieve the objective of a virtualized predictable and composable platform. CoMik creates a VP for every application. All the resources that are involved in the execution of an application on the platform are virtualized. Figure 3.1 presents an abstract view on the virtualization architecture. In order to create a virtual platform, different techniques have to be applied to every resources. For this purpose, in general, we use either of two techniques: (1) dedicating the resource, or (2) partitioning the resource.

Dedicating a resource to a VP means that the resource is assigned to a specific VP, and only the application executing on that VP is allowed to use this resource. In this work, such technique is only applied to RDMA modules in the processor tiles. As explained in Section 2.3.1, every RDMA is exclusively assigned to one application that executes in a VP.

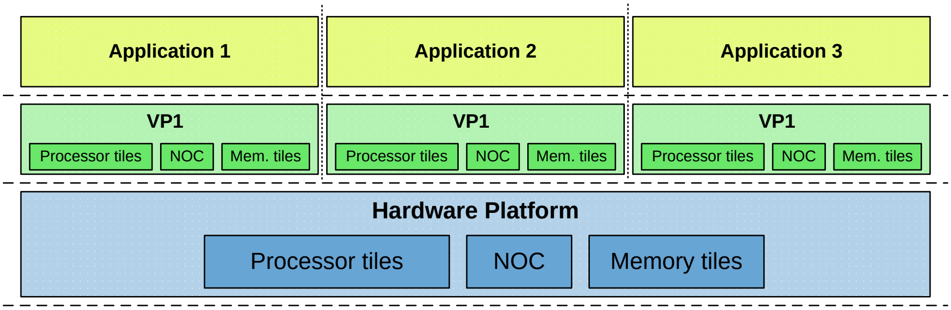


Figure 3.1: *An overview on the virtualization scheme of the CompSOC platform.*

All the other resources are virtualized using the partitioning technique. In the rest of this chapter[1] we first explain how the partitioning technique is applied to different resources of the platform, and we then describe the concept of composable VPs in the CompSOC platform. Following these, we present the implementation details of the CoMik microkernel. We motivate the needs of hardware support for virtualization where we detail the design architecture of TIFU[2]. Using this new hardware, we demonstrate how the interrupts are virtualized by CoMik. Before introducing how CoMik deals with critical sections, the interrupt handling routines are explained. Finally, the boot loading procedure of CoMik is presented.

3.1 Partitioning for Virtualization

Depending on the service type provided by a resource to its requesters, the resource can be partitioned *temporally* or *spatially* [65]. Temporal partitioning is applied to the resources whose utilization time is shared between a number of requesters, for instance, interconnect or processor. Spatial partitioning is typically applied to memory resources, for instance, the data memory of a processor. In this section, we discuss how the resources of target platform are partitioned either temporally or spatially.

In case of temporal partitioning of a resource, an arbitration scheme has to be used to manage the access time of the requesters to the resource, and therefore, to create a partition for every requester. A predictable and composable system imposes specific constraints on the arbitration schemes. In a predictable system, an arbitration scheme has to guarantee a minimum service provided by the resources to the requesters with real-time requirements. Moreover, in a composable system, the arbitration scheme has to also provide a guarantee on when the service is exactly available to a requester in order to prevent the interference of other requesters. TDM is one of the arbitration schemes that can be used for such a purpose.

TDM creates a set of time slots as fixed resource utilization time quanta. A temporal

[1] The content of this chapter is partially based on the following publications from the author of this dissertation and his colleagues:[6, 21, 23, 66, 71, 72, 77]

[2] This module replaces the CCM and PIT modules existing in the CompSOC processor tile architecture previously.

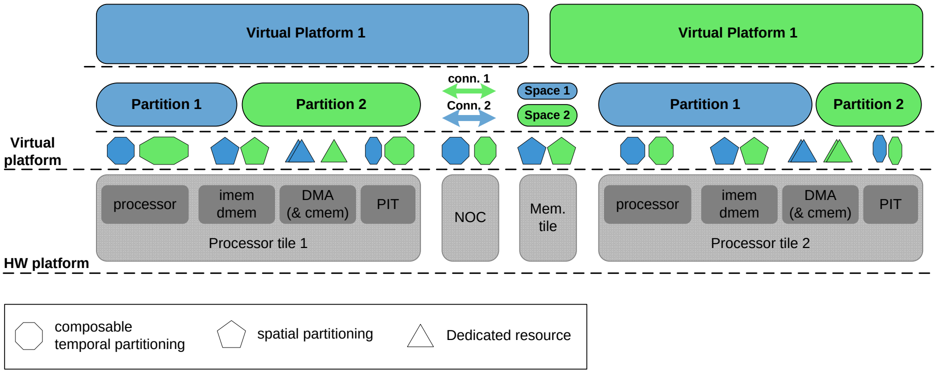


Figure 3.2: *An overview on applying partitioning techniques on the resources of the CompSOC platform to create VPs.*

partition is then a set of these time slots allocated to a requester following a periodic allocation table. We define the size of a partition as the number of allocated slots in a period of the table. Since the allocation table is fixed during the life time of a requester, a minimum service of the resource and the exact time that the resource is available to the requester can be guaranteed. In what follows, we discuss how this scheme is applied to the resources of our predictable and composable platform, when it is needed.

Assuming that we are going to execute two applications mapped on a two-tile instance of our target platform, Figure 3.2 presents all the hardware resources that might be involved in executing these applications. The techniques that are used to partition every individual resource in the granularity of their service units are visualized in the partitioning layer of the figure.

In the processor tile, we use TDM to create composable temporal partitions of the processor. In this way, we realize cycle-accurate temporal isolation between the applications. In Figure 3.2, two temporal partitions, i.e., green and blue, are created for two applications. An example of the TDM table for a platform accommodating these two partitions is illustrated in Figure 3.3. Every time slot is assigned to one partition, and each slot is further split into two fixed sub-slots. The first sub-slot in which the system software, e.g., OS, VMM, or hypervisor, executes the system services such as scheduling the applications, is denoted as the *system slot*. In the subsequent interval, denoted as the *partition slot*, the application code executes. Later in this chapter, where we introduce our software architecture, we discuss the implementation details of how temporal partitions are created on a processor.

The system time progresses based on the wall clock. However, for an application executing inside a partition, the time progress is seen differently. In a partition's slots, the time progresses normally and follows the system time. When a partition is swapped out with another partition, the time progress from this partition's perspective is being paused until the time that next slot of the partition is active. Then, the partition time resumes from the exact moment that has been paused in its previous slot. As illustrated in Figure 3.3, such a paused-resumed progress in system time for the partitions could be seen as a continuous *virtual time* progress. This time virtualization is implemented by

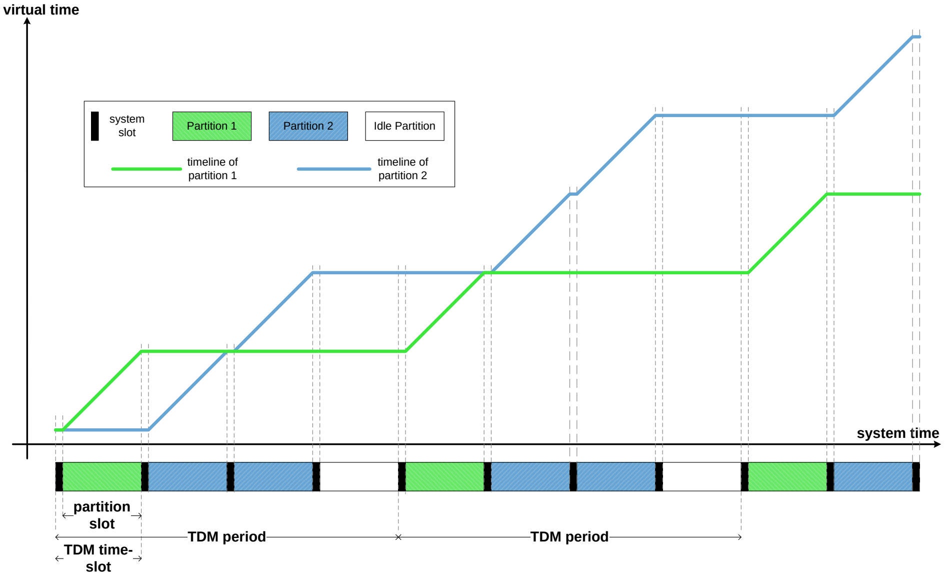


***Figure 3.3:** A TDM-based processor partitioning technique illustrating virtual time-line of two partitions.*

applying partitioning techniques to the timer module available in the processor tile of the platform. This technique is explained later in this chapter.

The instruction and the data memory, i.e., *imem* and *dmem*, can be partitioned to allocate dedicated memory regions to every partition code and the system software that execute on a processor tile. Practically, partitioning the *imem* is not necessary in our system, since all the system and applications code are compiled together at design time and are statically put in the instruction memory. Instead, the dynamic partition of *dmem* is created for every application at run time to be used for stack and scratch-pad memories. Figure 3.2 illustrates three spatial partitions of the processor tiles' *dmem*s, that are created for application 1 and application 2, colored in blue, and green, respectively. The same spatial partitioning could be also done for the shared memory tiles.

Each RDMA module with its associated memory is dedicated to one application. The number of these modules allocated to one application depends on the application's communication requirements.

In the discussion of the interconnect architecture in Section 2.3.1, we explained that the predictable and composable communication is provided at the granularity of the connections. A TDM scheme is then used to arbitrate the connections each of which belongs to one application [89].

Finally, the memory tiles are partitioned spatially and temporally. Spatial partitioning is applied for the back-end memory block, whereas the temporal partitioning technique is used for the front-end memory controller unit, as described in the literature [5, 24].

The following section explains how these partitioning techniques are used to create a VP for every application.

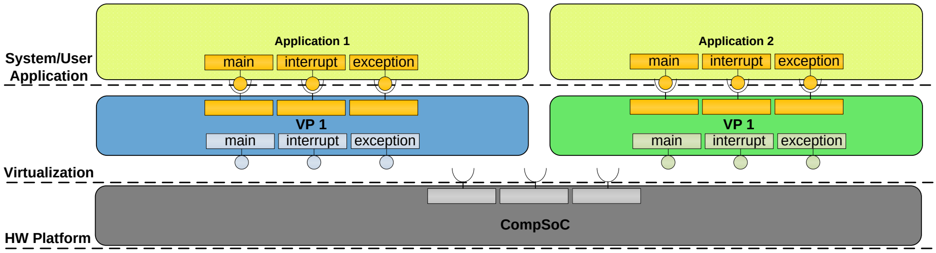


Figure 3.4: *Applications running on their dedicated virtual platforms.*

3.2 Composable Virtual Platforms

Figure 3.2 presents partitioning of all the resources. A processor tile partition is formed by a composable temporal processor partition, an spatial *dmem* partition, a number of dedicated RDMA modules, and a temporal PIT partition. A logical connection between two tiles is a temporal partition of the NOC, and finally, a dedicated memory block to an application is an spatial and temporal partition of the memory-tile.

A hardware architecture of a VP is logically defined as a set of resource partitions allocated to one application. In this way, every VP provides the application with an illusion of a dedicated actual physical hardware architecture. This is the first step towards the virtualization of an execution platform. The second step is to virtualize the execution of tasks' bodies of an application on each VP.

Typically, to execute an application directly on bare hardware, three instruction routines of the application have to be executed by a processor. These routines, which we denote as the *software hooks*, are: (1) *main* function, (2) *interrupt handler*, and (3) *exception handler*. The main function is the routine that implements the primary functionality of the application. The interrupt handler is the routine that is executed whenever the processor receives an interrupt, and similarly, the exception handler is the code that is executed when an exception (e.g., out of stack memory access) is raised. Every application is therefore expected to provide these software hooks to the processor. Consequently, as illustrated in Figure 3.4, every VP also provides a logical equivalence of these hooks so that the application feels no difference when comparing its execution in the VP to the actual bare platform.

In the rest of this chapter, we present the system software architecture that implements all the aforementioned partitioning techniques for creating VPs on the processor tiles, and executes the applications by providing virtual software hooks.

3.3 CoMik: a Composable Partitioning Microkernel

In the context of a predictable and composable system, the minimum services provided by a microkernel are distinguished as, (1) to create, control and schedule processor-tile partitions, (2) to execute an application in its partition by virtualizing the software hooks (i.e., the main function, the interrupt handler, and the exception handler), and (3) to

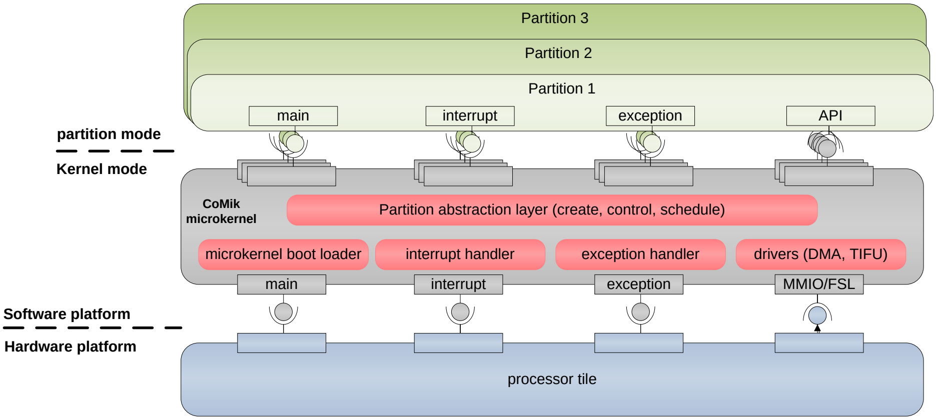


Figure 3.5: *The architecture of the software platform: CoMik in kernel mode, and partition routines in partition mode.*

provide an Application Programming Interface (API) so that the applications could use its allocated resources, for example, data send/receive API for using RDMA modules for communication purposes.

In order to implement such services in the CompSOC platform, we develop CoMik microkernel as illustrated in Figure 3.5. Every processor tile of the system executes one CoMik instance. On each processor, CoMik creates, controls, and schedules a number of partitions each of which is allocated to one application.

CoMik runs in *kernel mode* as it is a trusted software code with unrestricted resource access. The application software routines that execute in a partition, run in a *partition mode* as they are basically untrusted code and their accesses to the underlying resources are controlled and restricted by the microkernel. These application software routines may implement a user application or a guest OS. The focus of this chapter is on the structure of CoMik and generic partition-mode software architecture. Later Chapter 4 details the architecture of the software that executes inside a partition.

The boot loader of CoMik is the *main* function that a processor starts executing at the system start-up. Depending on the number of applications mapped to a processor at design time, CoMik creates and initializes a number of partitions. This is done by using the partition abstraction layer, as depicted in Figure 3.5.

Initially a CoMik Control Block (CCB) is instantiated on the processor. A number of Partition Control Blocks (PCBs) are created and linked to the CCB. Every PCB is associated to a partition and is linked with the partition's main function, interrupt handler, exception handler, and all control blocks of the temporal and spatial partitions of the other resources (e.g., a number of RDMA control blocks) allocated to this processor-tile partition. Figure 3.6 presents an overview of the CoMik control block structure. The temporal and spatial partitions of the processor-tile resources are statically created and linked to PCBs.

After creating partitions, CoMik uses these control blocks to schedule and control each application executing in the partition. Moreover, by wrapping the software drivers

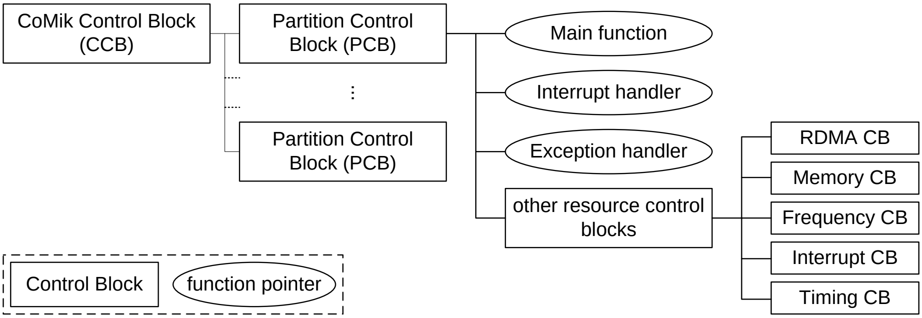


Figure 3.6: *Data structure of CoMik.*

of virtualized hardware resources, CoMik provides a set of API to the applications so that the applications can access to the resources. The API ensure controlled and restricted access of every application to its own temporal/spatial partitions of the resources. For example, each application can only allocate/free memory in its *dmem* spatial partition and it cannot affect the memory usage of allocation of other applications.

In the rest of this section, we first describe how CoMik realizes and schedules the temporal partitions. Second, we introduce the data memory partitioning layout implemented by CoMik. Following that, the extra hardware support required for our purpose of virtualization is introduced, and we explain the techniques that CoMik uses to virtualize the interrupts and exceptions. Finally, the general execution flow of CoMik is presented.

3.3.1 Composable Temporal Partitioning

The TDM-based technique for temporal partitioning of the processor is described in Section 3.1, and illustrated in Figure 3.7. Since CoMik plays the role of a VMM in our system, we denote the system slot as the *CoMik slot* in the rest of this thesis. CoMik implements the TDM technique using a periodic timer interrupt, denoted as the *system interrupt*, that signals the end of the partition slots. For this purpose, CoMik programs the PIT module with the fixed value of CoMik-slot size plus partition-slot size. More important, every partition slot has to start exactly periodically. The start time of a partition slot time is immediately after the CoMik slot that comes after the interrupt time of the last partition-slot's end, as illustrated in Figure 3.7. This implies that every CoMik slot has to also end at an exact moment in time. This is non trivial for CoMik to implement such a fixed slot due to the two following phenomena.

First, when a system interrupt is raised in the middle of executing either an uninterruptible critical region of the partition (application) routine or a multi-cycle instruction, the interrupt may be prevented from being handled immediately by the microkernel. This causes a variable jitter in the start time of the microkernel operations (t_a in Figure 3.7), i.e., scheduling and swapping the partitions, that should be performed in the CoMik slot. Second, these kernel operations may take a variable number of execution cycles from one slot to another, and consequently an undesired jitter in the start of the next partition slot appears (t_b in Figure 3.7).

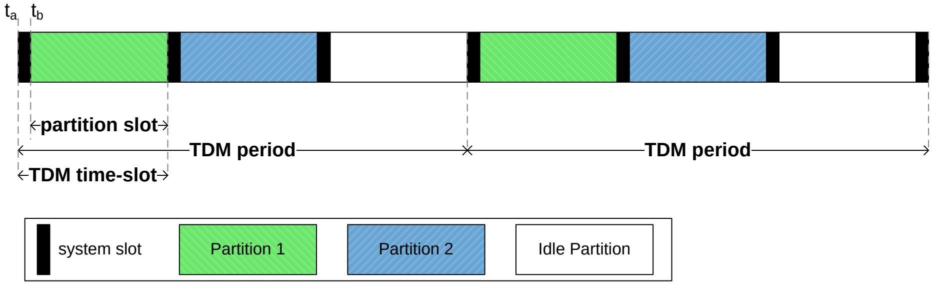


Figure 3.7: *A TDM-based processor partitioning technique.*

The proposed technique is to implement the CoMik slot by taking always into account the worst case of the both jitters. Figure 3.8 illustrates the details of this technique, given the fixed size of the CoMik slot is C time units. Assuming that the system interrupt notifies the end of the previous partition slot at time t in an uninterruptible section (of at maximum U_{wc} time units), it takes an actual delay of U_{ac} ($\leq U_{wc}$) time units for the processor to start handling the interrupt. In order to remove the undesirable effect of such a jitter, when the interrupt is handled at $t + U_{ac}$ time, the processor is halted (by clock gating) until the time of $t + U_{wc}$, where U_{wc} is the worst-case bound on the jitter and it is given at design time. Thus, the microkernel operations always start at a fixed moment in time.

Similarly, as illustrated in Figure 3.8, at the end of partition scheduling and swapping which take K_{ac} time units, the processor is halted until the time $t + (C - R)$. Therefore, the following condition always holds true: $U_{wc} + K_{wc} = (C - R)$. Here, R is the time that is needed to switch back to the next partition stack and load the processor context of the partition plus the time needed to preload the instructions of the next partition routine in the processor pipeline to compensate for the pipeline flushing done last time when this partition is interrupted. The stack switching and loading the context is implemented with the fixed execution time, and in case of the Microblaze processor, in the CompSOC platform, preloading part takes just 2 cycles.

Since the CoMik slot is generally an overhead of partitioning, it is desirable to minimizes its size. Based on the previous formalization, two approaches can be taken to minimize the CoMik overhead: (i) minimize K_{wc} by efficient implementation of CoMik code, and (ii) minimize U_{wc} by reducing maximum size of the critical (uninterruptible) application code section.

3.3.2 Scheduling & Swapping Partitions

As illustrated in Figure 3.8, the main operations performed by CoMik in its slot are scheduling and swapping the partitions. As each partition corresponds to an application, basically the partition scheduling implements preemptive, time-triggered inter-application scheduling as described in Section 2.4.

Swapping the partitions is done by swapping out the running partition, and then scheduling the next partition. First, the context of the interrupted partition is stored.

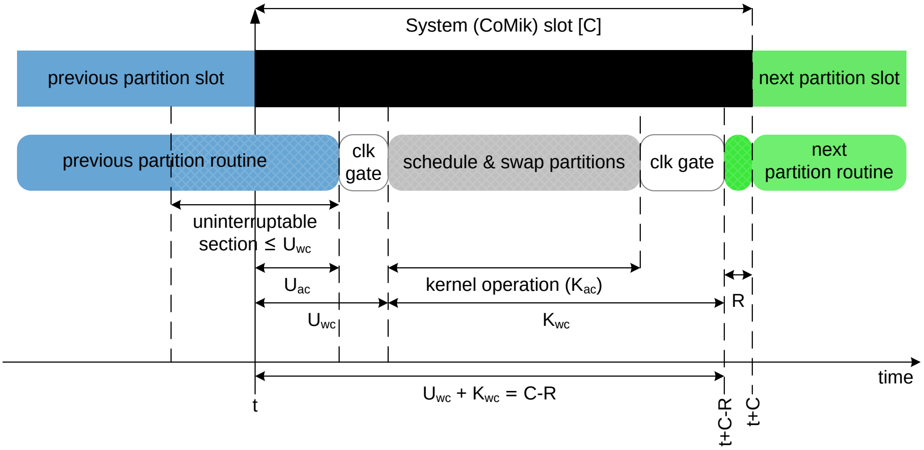


Figure 3.8: A detailed view on the kernel operations and timeline in a CoMik slot.

The context of a partition includes the processor registers, and possibly the status of other hardware resources that are shared between the partitions. An example of such resources is the TIFU that we introduce later in this chapter. The processor registers are stored on the dedicated stack space of the partition, while the rest of the context is kept in the control blocks associated with the partition's PCB.

In the partition scheduling step, the next partition is selected according to the TDM table. If a TDM slot is not allocated to any application in the table, the idle partition is chosen. When the idle partition is scheduled, the processor is halted for the duration of the partition slot to reduce energy consumption. Afterward, the context of the next partition is read from its PCB and loaded from its stack, the corresponding hardware resources are programmed and the the processor's registers are set.

3.3.3 Memory Partitioning Layout

The data memory (i.e., *dmem*) is statically divided into three logical sections by the linker script[3] at design time: (1) global data, (2) system *heap*, and (3) system *stack*, as illustrated in Figure 3.9. The first section is reserved to hold the global and constant variables of the compiled code including the CoMik code. The system heap and stack sections are used and managed by CoMik at run time.

Every partition requires isolated heap and stack memory sections. For this purpose, as demonstrated in Figure 3.9, CoMik allocates a fixed amount of memory space for every partition's private heap and stack by partitioning the system heap dynamically at run time. The idle partition (Partition 0) does not need any heap memory, but since it may be interrupted, a very small stack section is required.

The size of allocated heap and stack memory to each partition can be different from

[3] A linker script describes the memory layout of the target machine, and specifies where each program section should be placed in memory.

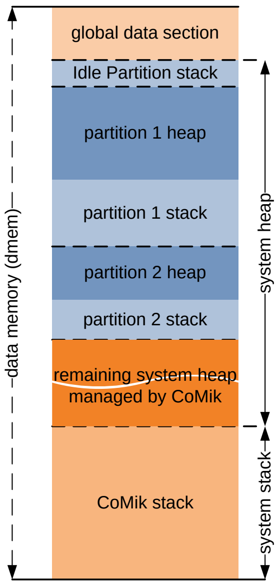


Figure 3.9: *The data memory partitioning layout.*

the others' depending on the routines executing in the corresponding partitions. Thus, the memory control blocks linked to each PCB has to keep the start, the end, the size, and the current pointer of heap and stack sections.

The CoMik code uses exclusively the system stack (known therefore as CoMik stack) and the remaining part of the system heap, while executing the main function, interrupt handler and exception handler.

3.4 A Hardware Support for Virtualization

The hardware architecture of the CompSOC processor tiles has to assist CoMik with the basic functionality required to virtualize the platform. Timer-based functionality is an essential required feature for this purpose. This motivates us to develop TIFU as a hardware module that assist CoMik for virtualizing the platform. Figure 3.10 presents the architecture of the processor tile including TIFU. TIFU interfaces with the processor via FSL, generates tile clock, and issues an interrupt signal to the processor.

In the rest of this section, we explain the functions and detailed architecture of TIFU, as presented in Figure 3.11. TIFU has a modular architecture and is built around two main clock counters that trigger three timers and a frequency controller module. The interrupt control module receives all the timer interrupts and the external interrupts to signify the processor via the interrupt line. CoMik manages all these modules via the control unit.

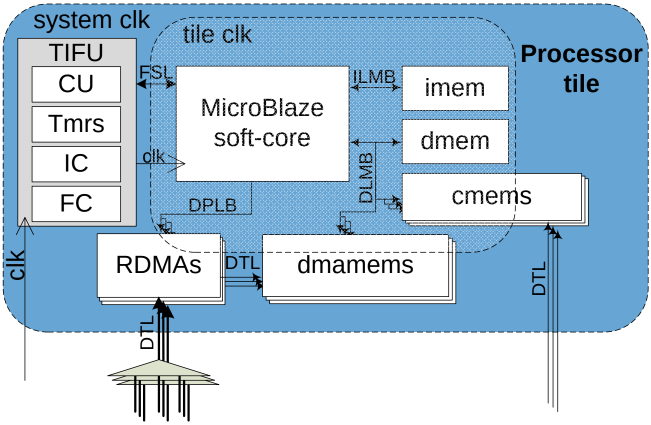


Figure 3.10: The processor tile architecture including TIFU.

3.4.1 Counters

A counter is a module that increments a register value based on clock ticks. The TIFU contains two counter modules: (1) *system counter*, (2) *tile counter*. As the names suggest, the system counter works on the system clock, while the tile counter works on the tile clock which is a scaled version of the system clock and is generated by the frequency controller module. The both counters are implemented with 64-bit registers to prevent an overflow of the counters in a reasonable lifetime of the system.

The tile counter keeps track of the executing partition's virtual time. When a system interrupt signifies the end of the corresponding partition slot as illustrated in Figure 3.8, the tile counter is stopped and its register value is stored as part of the hardware context of the previous partition. Later, the counter is loaded with the last value of the next partition from when it has been stopped and the counter resumes from this value at the start of the next partition slot. For this purpose, CoMik is able to get/set the register values of the both counters.

3.4.2 Timers

A timer is a module that generates timed interrupts based on a time reference. The TIFU is equipped with consists of three timer modules: (1) the *system timer* that generates system interrupts based on the system counter, (2) the *tile timer* that generates interrupts based on the tile counter, and (3) the *mix timer* which is exactly the same as the system timer but generates interrupts to be used inside the partitions. The rationale behind the mix timer is that if an application needs to use a timer based of the system counter, it must not interfere with the functionality that is needed to create temporal partitions by the system timer.

A timer interrupt is generated by comparing the value of a programmable register inside the timer with the value of the corresponding counter. In order to set this register, the timer can be programmed in either of two methods: (1) an absolute value of the time

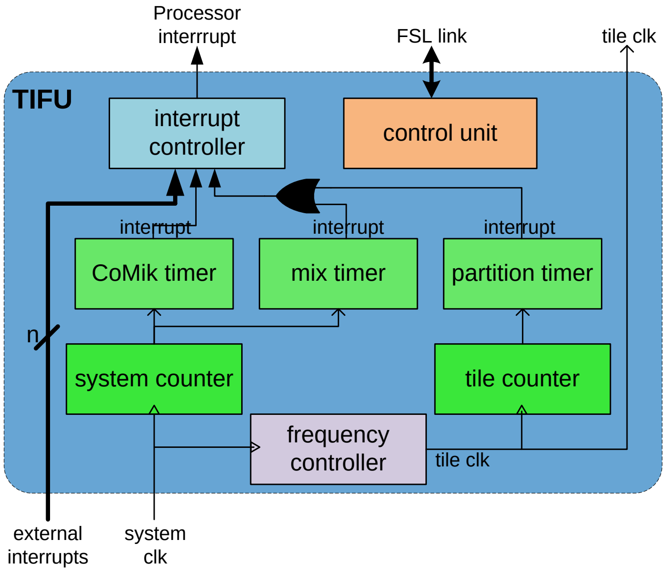


Figure 3.11: *The TIFU architecture.*

moment at which the interrupt has to be generated is given, (2) a relative value of the time moment with respect to the current value of the corresponding counter is given. In the second method, the timer automatically adds up the relative value to the current value of the counter and sets the programmable register.

Furthermore, the timers are designed to generate interrupts in two different modes: one-shot and periodic mode. In the first mode, the timer is programmed with a time moment to generate an interrupt once. In the second mode, the timer is programmed with a time duration to generate a sequence of interrupts periodically after an already programmed interrupt.

The tile and mix timers can be set in one-shot or periodic mode by the routines of an executing partition, while CoMik always programs the system timer in the periodic mode to implement TDM-based temporal partitions. When swapping the partition in a CoMik slot, the register values of the tile and mix timers are stored/loaded as part of the hardware context of a partition. For this purpose, CoMik can get/set the programmable register values of the timers.

3.4.3 Interrupt Controller

The interrupts that a processor may react to in the CompSOC platform are categorized from the perspective of TIFU as internal and external interrupts, depending on the source of an interrupt. Internal interrupt sources that generate timed interrupts, are the three timers explained previously in this section. The system timer-interrupt is used by CoMik to implement temporal partitions. The mix and the partition timer, each generates an interrupt. These interrupts are ORed together in a signal called *partition timer-interrupt*, as illustrated in Figure 3.11. An external interrupt source is either an in-tile or off-tile resource intending to notify the processor that an event has been occurred. For instance, an external I/O module.

Inside TIFU, the internal and the external interrupts are registered by the interrupt controller in an *interrupt vector register*. Since the processor accepts an interrupt via the one interrupt line, whenever any of the internal and external interrupts are raised and a bit in the interrupt vector is set, the processor gets signified through that line.

Moreover, in partition time, the processor does not always have to react to all the interrupts forming the interrupt vector register. In order to ignore some of the interrupts that do not belong to the executing partition, the interrupt controller is equipped with a masking register. When a bit is set in this register means that the processor has to handle the interrupt linked with the corresponding bit in the interrupt vector register. This is implemented by ANDing the masking and the interrupt vector register to create a *masked interrupt vector register*. The processor interrupt line is then sensitive to this register.

When the interrupt line of the processor is set, the source of the interrupt has to be identified. For this purpose, CoMik has to access the registers. Additionally, when swapping the partition in a CoMik slot, the masking and the interrupt vector registers are stored/loaded as part of the TIFU context of a partition. Thus, CoMik can get/set all these registers.

3.4.4 Frequency Controller

The frequency controller is implemented in TIFU, following a clock division technique, as illustrated in Figure 3.11. The TIFU receives the system clock as an input and based on the given scaling ratio generates the tile clock with a frequency lower or equal than the system frequency. The frequency change of the tile clock may be programmed in two different modes, as one-shot or interval.

In the first mode, the frequency change is done for either an absolute or relative time in the future with respect to the system or tile counter time references. In the second mode, the frequency is changed for a duration of time, and after the interval is finished, the tile frequency is changed back to the original one. The duration can be expressed by giving a relative value to the current time, or by giving an end time in the future.

Using these frequency change modes, the CoMik and any of the partitions routines may run with different clock frequencies. In general, to reduce the overhead of CoMik slot, the code executing in this time runs on highest possible frequency of the tile clock, which is equal to the system frequency. For this purpose, the frequency controller receives the system interrupt so that when the interrupt is raised and a CoMik slot starts, it increases the tile frequency.

Inside a CoMik slot, the interval frequency change mode is used to implement the processor halting by changing the tile frequency to zero until the given moment in time (as explained in Section 3.3.2).

The programmed frequency changes and the state of the frequency controller are registered as part of the TIFU context, and CoMik can store/load them when swapping the partitions in its slot. Thus, when a CoMik slot ends, the next partition starts with the tile frequency that had been running on the last time that the partition was swapped out.

3.4.5 Control Unit

The control unit of TIFU is designed following an instruction-based architecture. The processor interfaces with TIFU by giving the instructions to the control unit via an FSL. The control unit is then decodes the instructions and forwards them to the target modules.

The instruction are generally categorized in two sets of set and get instructions, which are used to access the register values of the counters, the timers, the interrupt controller, and the frequency controller module. All the instructions are provided as part of a software driver to be used by CoMik, as illustrated in Figure 3.5.

3.5 Interrupt Virtualization

The interrupts are categorized generally as system- and partition-level interrupts. The system-level interrupt targets the microkernel. The only example of such an interrupt is the system timer interrupt that is handled by CoMik to implement temporal partitions and realize preemptive time-triggered partition (inter-application) scheduling.

The partition-level interrupts however target a specific partition. When a partition level interrupt occurs, it has to be handled in its corresponding partition time, composably. This means that not only the other partitions are not notified that any interrupt has been occurred, but also handling the interrupt does not affect their temporal and functional behavior. An example of such an interrupt is when an application executing in a partition uses a timer interrupt to implement preemptive time-triggered intra-application scheduler.

The TIFU supports partition-level interrupts including the non-timer ones. Interrupts virtualization in this approach is not the matter of co-existing partition and system interrupts, but it enables CoMik to virtualize partition-level interrupts between the partitions.

All the partitions may use mix and tile timers. When the programmable and status registers of the timers, together with the tile counter are stored/loaded in the CoMik slot, as the partitions gets swapped. Thus, every partition uses mix and tile timers in full isolation with the other partitions.

The timer and external interrupts are registered in the interrupt vector by the interrupt controller. At the partition time, it is the interrupt mask vector that determines to which interrupt a partition has to react to. Every partition has its own mask vector which can change dynamically at run time independently from the other partitions. The only interrupt that is not mask-able by the partition-mode code is the system interrupt to prevent invalidating the system composability.

3.6 Interrupt Management in CoMik

When the processor is signified via the interrupt line that an interrupt has been occurred, the running application routine (if it is not in a critical section, otherwise after such a section) is preempted and the CoMik interrupt handler receives the control of the processor.

Depending on the interrupt source(s), CoMik has to distinguish between the system and partition interrupts. The system interrupt is handled in the CoMik interrupt handler, while the partition interrupts are handled by the interrupt handler of the running partition. In what follows, we describe the execution flow of handling system and partition interrupts, in order.

3.6.1 CoMik Interrupt Handler

Figure 3.12 illustrates the CoMik interrupt handling flow. When the running application is interrupted, the handling routine starts storing the processor context on the stack memory of the interrupted partition. In order to be able to load this context later, the pointer to the interrupted stack memory is kept in the CCB, and the working stack is switched to the CoMik stack.

In order to find out the source of the interrupt, the masked interrupt vector is read from TIFU interrupt controller. If the system timer is the source of the interrupt, a CoMik slot has to be implemented as illustrated in Figure 3.8. Note that, if the partition and the system interrupts are both set in the interrupt vector, the precedence is always given to handling the system interrupt, since it implements the CoMik slot. Following the description in Section 3.3.2, the interrupt handler then implements the partition scheduling and swapping.

The handler first stops the partition timer and reverts the value back to the exact time of the end of the previous partition. Following that, the system interrupt is acknowledged by resetting the corresponding signal in the hardware.

The variable jitter in start of the CoMik slot is removed by halting the processor until an specific moment in time. When the processor wakes up, the current context of TIFU for the interrupted partition is stored, and the TIFU is checked for errors.

Before scheduling the next partition based on the TDM table, we switch the working heap space to CoMik's heap. After the next partition is scheduled: first the TIFU context of the partition is loaded; second, we switch the heap memory to the partition's heap; and third, the jitter in scheduling operations is removed by halting the processor again until an specific moment in time.

Finally, when the processor wakes up, we switch back to the next partition stack and load the processor context of the partition. After preloading the instructions of the next partition routine in the processor pipeline to compensate for the pipeline flushing done last time when this partition is interrupted, at the end of the interrupt handler the control returns to the next partition routine. If it is the first time that the partition is scheduled, the control returns to the main function of the partition.

When the processor receives an interrupt signal and after it enters the interrupt handler the processor interrupt is disabled automatically by setting the processor status register. The interrupt remains disabled during the handler execution, and at the end when the control returns from the interrupt handler, the interrupt is enabled automatically.

3.6.2 Partition Interrupt Handler

Every partition has a partition interrupt handler. When the CoMik handler finds out that the interrupt is a partition one, the corresponding partition interrupt handler has to be called. We support two types of the partition interrupt handlers in CoMik, integrated and standalone.

The integrated partition interrupt handler relies on the fact that the part of the CoMik handler has been executed before, and it will eventually return back to the CoMik handler. Instead, the standalone interrupt handler is totally independent from what CoMik handler has done before, and it will never come back to it.

In what follows, we explain how CoMik handler passes the control to each of these two types of partition interrupt handlers, as depicted in Figure 3.12.

Integrated Handler

An integrated partition interrupt handler is developed with the assumption that the interrupted context has been already stored on the corresponding partition stack by the CoMik interrupt handler. Thus, maximum size of the corresponding application's critical section has to be taken into account to estimate the worst-case jitter bound of the CoMik slot, i.e., U_{wc} illustrated in Figure 3.8.

In order to call such a partition interrupt handler, first, CoMik switches back to the partition working stack. It then disables the partition interrupts by changing the interrupt mask vector in the TIFU, and it enables the processor interrupt so that the system interrupt is not going to be missed. After this, the partition interrupt handler is called.

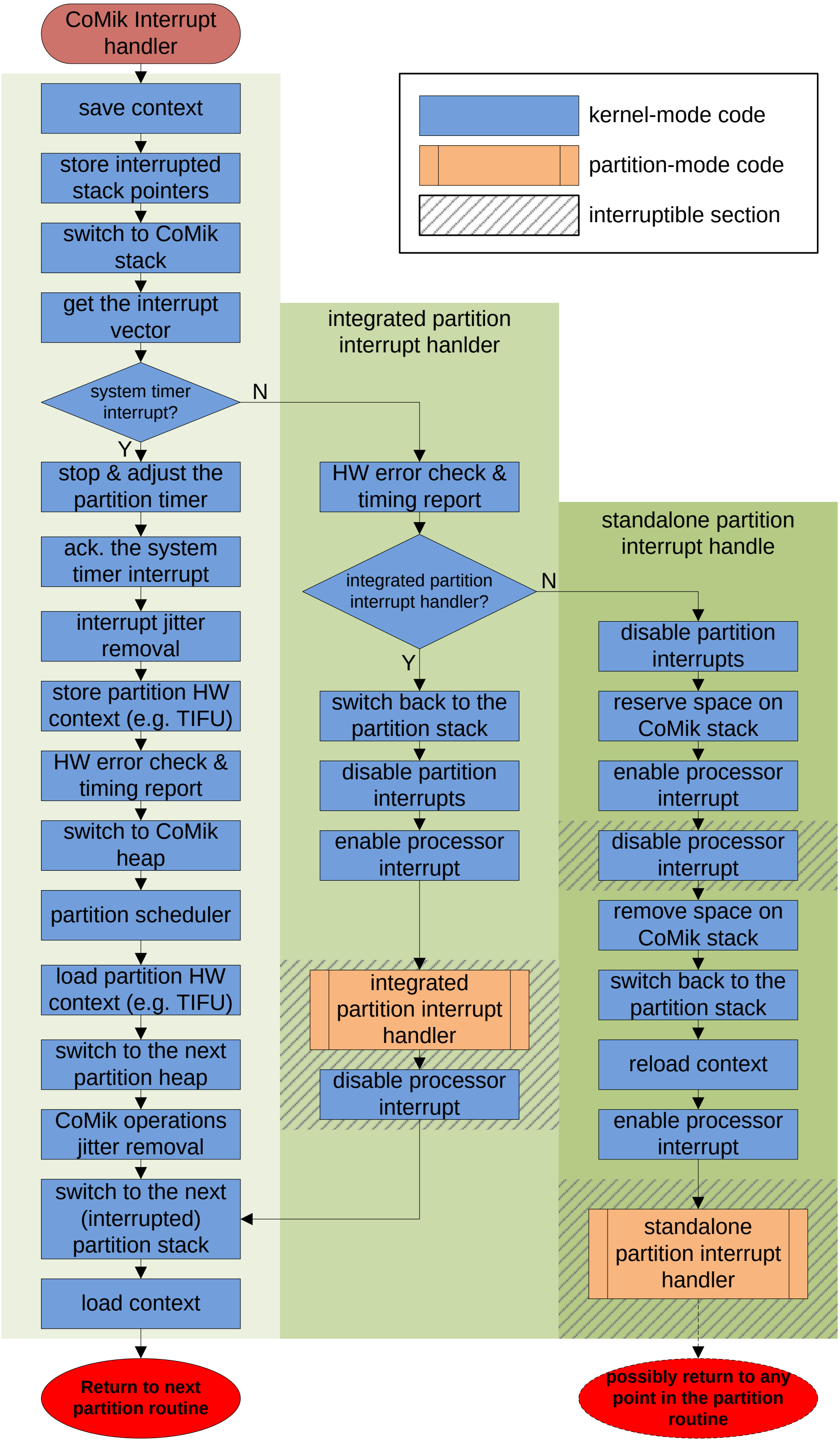
This type of the interrupt handler is implemented and called like a function, meaning that it has to return back to the callee which is CoMik in this case. After it has returned, the processor interrupt is immediately disabled so that the loading context of the interrupted stack is protected. The rest of the operations are done as explained for the system interrupt handler, and it is depicted in Figure 3.12.

Standalone Handler

An standalone interrupt handler is developed totally independent of what the CoMik handler has been doing before its call. It therefore does not contribute to any worst-case jitter bound of the CoMik slot.

As a typical interrupt handler takes care of storing and loading the interrupted context, the standalone handler prepares a fully virtualized environment for such an interrupt handler used by an application in a partition.

In order to call the standalone type partition interrupt handlers, the processor context has to be reverted back to the same state as it was when it has been interrupted. For



CoMik Interrupt handler
save context
store interrupted stack pointers
switch to CoMik stack
get the interrupt vector
system timer interrupt?
N
Y
stop & adjust the partition timer
ack. the system timer interrupt
interrupt jitter removal
store partition HW context (e.g. TIFU)
HW error check & timing report
switch to CoMik heap
partition scheduler
load partition HW context (e.g. TIFU)
switch to the next partition heap
CoMik operations jitter removal
switch to the next (interrupted) partition stack
load context
Return to next partition routine
kernel-mode code
partition-mode code
interruptible section
integrated partition interrupt hanlder
HW error check & timing report
integrated partition interrupt handler?
N
Y
switch back to the partition stack
disable partition interrupts
enable processor interrupt
integrated partition interrupt handler
disable processor interrupt
standalone partition interrupt handle
disable partition interrupts
reserve space on CoMik stack
enable processor interrupt
disable processor interrupt
remove space on CoMik stack
switch back to the partition stack
reload context
enable processor interrupt
standalone partition interrupt handler
possibly return to any point in the partition routine

Figure 3.12: *CoMik interrupt handling flow.*

this purpose, as illustrated in Figure 3.12, the stack is switched back to the partition stack and the context that has been stored at the beginning of the CoMik interrupt handler, is loaded. But, before this, the partition interrupt has to be disabled by changing the interrupt mask vector.

As the processor interrupt is still disabled at this stage, we enable and disable the interrupt immediately so that to break having a long critical section that the system interrupt cannot react to. We again disable the processor interrupt just before calling the standalone partition interrupt handler.

When the control is passed to the partition interrupt handler, the processor context is exactly the same as the moment it has been initially interrupted, but, the processor interrupt is enabled and only the partition interrupts are disabled. This partition interrupt handler may then do whatever it needs to without any obligation to return back to any specific routine.

3.6.3 Exception Management in CoMik

The procedure of handling an exception is very similar to the one of the interrupts. The CoMik exception handler is the first entry point of the exception, it is to identify how to handle each exception. If an exception has to be handled by the partition particularly, the partition exception handler is called; otherwise, the CoMik handles the exception.

Currently, all the exceptions are handled by the CoMik in the way that it first reports the exception that has been occurred and then it stops the running partition. The faulty partition is replaced with the idle partition in the TDM table so that it will not get scheduled in any of the future slots.

3.7 Critical Sections

The CompSOC platform defines a critical section as a piece of code that has to be executed uninterruptedly. An example of such a section is the TIFU driver API that sends two 32-bit values consecutively via FSL (which a 32-bit wide link) to program a 64-bit register in the hardware, and the code must not be interrupted in between sending these two values.

In order to implement a critical section, all or some of the interrupts have to be disabled for the duration of executing the section. For this purpose, there are two mechanisms available in the CompSOC platform: (1) disabling the processor interrupt using the processor status register so that the processor does not react to any interrupt, (2) disabling the interrupts by masking them in the interrupt mask register of TIFU so that it does not issue any interrupt to the processor. Depending on the mechanism that is used, we categorize the critical sections in two classes of kernel-mode and partition-mode. In the following, we discuss each of these two classes in detail.

3.7.1 Kernel-Mode Critical Sections

During the execution of a kernel-mode critical section, we disable the processor interrupt using the status register. The processor therefore is not signified by any of the system and partition interrupts.

Since the system interrupt implements the composable temporal partitions, any delay in handling this interrupt is acceptable only if it is less than or equal to the jitter-bound (U_{wc}) presented in Figure 3.8. Thus, a kernel-mode critical section is a fully trusted code with bounded execution time.

In order to implement such class of critical sections in the CompSOC platform, the Microblaze processor comes with an API to disable/enable the processor interrupt by manipulating its status register.

3.7.2 Partition-Mode critical sections

During the execution of a partition-mode critical section, we disable the interrupts by setting the corresponding bits in the interrupt mask register of TIFU. The partition-mode critical section is an application code that executes in a partition. Therefore it is not a trusted code because if it were to execute too long it must be possible to swap out the partition composably.

Using the interrupt mask register, both system and partition interrupt may be disabled. Disabling only the partition interrupts is not a big challenge, however, if the system interrupt is also disabled, there would be a significant challenge in enforcing the required execution time bound to the critical section, in order to not compromise the system composability. We therefore introduce two subclasses of partition-mode critical sections, as described in what follows.

System-interruptible

In this class of partition-mode critical sections, only the partition interrupts are masked in the interrupt mask register, and the code may be interrupted by the system interrupt. Examples of such a critical section are the (integrated and standalone) partition interrupt handlers described in Section 3.6.2.

In order to implement this class of critical sections, CoMik provides applications with an API to mask/unmask the interrupts by setting/resetting all the corresponding bits (to the running partition) in the interrupt mask register.

Non-interruptible

In this class of partition-mode critical sections, all the interrupt including the system interrupt are disabled. The mechanism of disabling the interrupts is similar to the one of the system-interruptible class.

Maximum size of such a critical section has to be less than the worst-case jitter bound of the CoMik slot, i.e., U_{wc} illustrated in Figure 3.8. To enforce the execution bound, a watchdog timer technique is implemented. In this technique, at the beginning of the critical section, when the interrupts gets disabled, the state of the mix timer is stored and it is programmed to perform as a watchdog timer with the value of the jitter bound. In a normal case, if the critical section ends before the watchdog fires, the mix timer is set back with its state as it was before the critical section. However, if the watchdog fires before the critical section ends, it means that the execution time of the section goes over the acceptable jitter bound and it can possibly invalidate the system composability. In this case, the interrupt that is issued by the watchdog is handled by the CoMik in such a way that the running partition is stopped, an error is reported, and the system continues without any problem. The faulty partition gets never scheduled back in the system in any of the following partition slots, and it is replaced with the idle partition in the TDM table.

3.8 CoMik Boot Loading

The process of booting up a virtual platform on the corresponding processor tiles starts by creating and initializing its partitions. In each processor tile, the CoMik main routine is the first entry point of execution when the system boots up. A flow diagram of this routine is presented in Figure 3.13. In general, this routine creates a CoMik instance on the processor tile, realizes the static temporal and spacial partitions of the resources, at run time, and starts scheduling and executing the partitions.

The CoMik main routine is logically composed of two sections, as depicted in Figure 3.13: (1) a boot loader which is a user-written section, and (2) a start-up which is CoMik trusted code. In the first section, the user can statically configure the partitions on the processor tile; following that, the start-up section actually starts executing the partitions.

At the beginning of the boot loader, the TDM table, where each slot is allocated to one partition, is initialized. The allocation is given at design time according to the requirements of an application running on each partition. Afterward, a CCB is created with the structure presented in Figure 3.6. The initialized TDM table is then linked to this CCB.

As every resource, such as the RDMAs, has its own control block that its driver operates on, these control blocks need to be initialized before they are actually used. For example, every RDMA control block is initialized with the memory-mapped address of the actual hardware resource. This has to be done before linking the resources to the partitions.

At the end of the boot loader, a number of partitions are created according to a design-time recipe. The recipe determines the number of partitions with their parameters and the size of a partition slot. Each partition is linked with three pointers to main, interrupt handling and exception handling routines. Moreover, based on the given required memory space, for every partition, a memory control block is created following the layout presented in Section 3.3.3.

After the boot loader section, in the start-up section, CoMik first initializes TIFU with the given parameters. It set the system timer with the value for an initial interrupt.

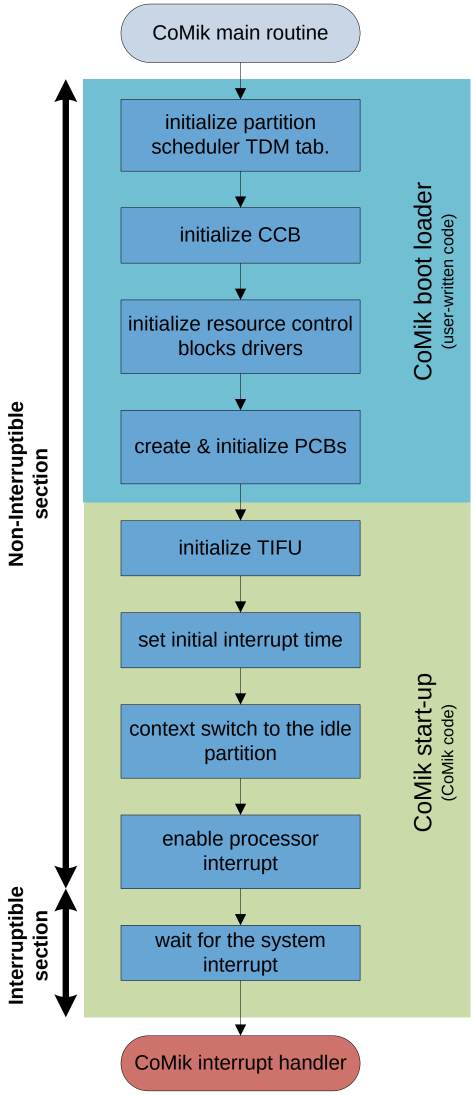

CoMik main routine
initialize partition scheduler TDM tab.
initialize CCB
initialize resource control blocks drivers
create & initialize PCBs
CoMik boot loader
(user-written code)
initialize TIFU
set initial interrupt time
context switch to the idle partition
enable processor interrupt
wait for the system interrupt
CoMik start-up
(CoMik code)
CoMik interrupt handler
Non-Interruptible section
Interruptible section

Figure 3.13: *Execution flow of the CoMik boot loader.*

The periodic scheduling and executing of the partitions starts after this initial interrupt. Before enabling the processor interrupt and waiting for the initial interrupt, CoMik first switches the context to idle partition's so hat once the interrupt occurs, the interrupt handler routine (as depicted in Figure 3.12) follows its normal execution flow.

3.9 Related Work

With respect to the context of this chapter, a large body of work has been published in using virtualization technology and partitioning technique to offer composability property for embedded systems [32]. This section gives an overview some of the most relevant efforts to our approach.

Virtualization technology has gained recently a lot of attentions as a design trend in embedded systems [1, 32, 41, 51]. To virtualize the resources they have to be either dedicatedly allocated to an application [54] or they have to be partitioned [55, 65, 69]. Depending on the service provided by a resource, it can be partitioned either temporally or spatially [65]. Two of the main standards in the domain of automotive and avionic applications are Aeronautical Radio Incorporated (ARINC) and AUTomotive Open System ARchitecture (AUTOSAR). RTOSs that are designed based on these standards are to support the partitioning of hardware resources in order to enable realization of VPs for mixed-criticality systems.

In the avionic domain, ARINC standard is a specification for time and space partitioning in mixed-criticality avionics RTOSs [83]. It specifies partitions at application level, where one or more applications with different criticality can belong to one partition [96]. The standard also specifies APIs for abstraction of the application from the underlying hardware and software platform. The RTOSs developed based on this standard guarantee the minimum amount of service that a partition receives. Thus, an application executing in a partition is affected by the presence/absence of other partitions.

In the automotive domain, AUTOSAR is a standard automotive software architecture, jointly developed by automobile manufacturers, suppliers and tool developers [9]. It contains an RTOSs specification which defines real-time performance, scheduling strategy and temporal partitioning for executing applications with mixed-criticality. In this standard the partitions are implemented with a schedule table and a time monitoring mechanism is used to limit the worst-case execution time of the applications. Thus, an application timing behavior can be possibly affected by the execution of other applications.

Typically, in a virtual platform, a hypervisor manages a number of VMM that run guest OSs. The scheduler of virtual machines may follow different policies and it is not fully decoupled from the scheduler inside the guest OSs. Specially in case of RTOSs, the two scheduling levels cooperate with each other to ensure the timing requirements of the time-critical applications are met [102]. In our composable approach, the function of the hypervisor is realized by CoMik, where instead of a guest OS, the applications can directly schedule their own tasks. This approach suits embedded systems as it avoids overhead of having guest OSs in terms of execution time and memory footprint, in the cost of losing high-level services that an OS could provide to applications. Moreover, many other VM-based embedded systems exist, especially in the automotive domain [49, 52, 63]. Among all, the approach in [49] is the closest to our technique as it also targets time-driven applications. It implements two-level scheduling, where in the first level, the hypervisor allocates a single time slot to every virtual machine, and at the second level, inside each virtual machine, a guest RTOS may use a preemptive scheduler. To the best of our knowledge, none of the existing work offer a fully composable platform (by our definition of composability) that executes time-triggered applications.

None of these aforementioned approaches and consequently the system compliant with them are fully comply with the definition of strong composability, and therefore are distinguished as weakly composable systems.

Other examples that employ temporal resource partitioning are presented in [69] and [55]. They execute multiple non-time-triggered applications concurrently, ensuring that they are not affecting each other's worst-case timing behavior. The authors of [69] utilize two-level scheduling, TDM inter-application and cooperative static-order intra-application. The work in [55] analyzes applications individually to enable reasoning about their overall worst-case behavior when executing them on the same platform. These works offer composability of worst-case bounds of applications, and therefore they can be categorized as *weakly* composable systems. This makes them suitable for systems containing only firm-real-time applications.

Our proposed approach is however strongly composable in the sense that, inter-application interference is completely prohibited at cycle-level. Thus the actual-case performance of applications is independent, enabling mixed-criticality applications to be designed, verified, integrated (without re-verification), and executed on the same platform.

The implication of strong composability for processor resources of a system is to be instruction level predictable so that the cycle-level isolation between the applications with mixed-criticality executing on the same processor is guaranteed. To achieve this, a large body of research work has been done [58, 60, 86, 103]. A common taken approach in all the existing work is that they propose a new or modified hardware architecture in order to enforce predictable timing of the processing elements and consequently make the isolation between applications composable. In contrary, our approach, as implemented by CoMik, is not really processor architecture dependent. Here, we use the technique of processor clock gating until an exact given moment in time, by which we create a time-predictable behavior of the processor. Although this approach does not require any hardware (processor) modifications, it comes with some limitations. Currently, some of these limitations are: (i) the processor has to not use any level of data/instruction caching mechanism, (ii) worst-case execution time of the code critical sections have to be bounded and known at design time, (iii) the processor cannot use branch prediction, and (iv) instructions out-of-order execution is not allowed.

In the context of interrupt virtualization, some existing approaches, such as uC-OS/II, multiplex multiple timed events on a single tick generator, i.e., a periodic interrupt timer. They therefore offer software-based virtual timers [38, 98]. In these approaches, the granularity of the virtual timed interrupts is restricted to the actual underlying hypervisor or RTOS ticks. Whereas in our approach, we virtualize a single timer interrupt line of the processor that is used by each application using the TIFU. Every application has access to the physical timer that access is exclusive. In each partition, an application can set its own timed events independent of the other partitions, and the granularity of the events can be wither the granularity of the system or the tile counter. CoMik takes care that interrupts of an application are always mapped to its temporal partitions, and never interfere with other partitions (and hence behavior) of other applications.

3.10 Summary

In this chapter, we use the virtualization technology for the purpose of creating a predictable and composable platform. This technology is realized by the CoMik microkernel, which creates a VP for every application by temporal and spacial partitioning of all the resources that are involved in the execution of the application on the CompSOC platform.

CHAPTER 4

Realization of the Model of Execution

THE model of execution fills the gap of execution abstraction between the models of computation's semantics and the primitive execution operations supported by the platform. In the context of the CompSOC platform, the CoMik microkernel creates a virtual platform for each application that is expressed in either of the KPN, the CSDF, or the time-driven model of computation. The model of execution is therefore an intermediate layer in the platform stack, as illustrated in Figure 4.1.

The model of execution is implemented in form of an OS, namely CompOSe, instantiated as library in every partition of the virtual platform created by CoMik for an application.

In this chapter[1], we first motivate the requirements of the model of execution and introduce a realization of such a model on CompSOC's virtual platforms. Second, we present the implementation of the model of execution by the CompOSe library and we detail the structure and architecture of the library. Here, we show how different models of computation can be developed to execute on the platform. Finally, we give an overview on the related work.

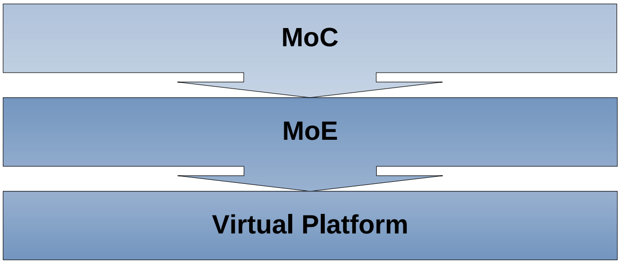


Figure 4.1: *CompSOC platform stack.*

[1] The content of this chapter is partially based on the following publications from the author of this dissertation and his colleagues:[66, 72–74]

4.1 Model of Execution

A model of execution defines a set of operations and their orchestrations in order to implement an specific model of computation. As discussed in Section 2.5, an application is typically expressed in a model of computation, e.g., KPN or CSDF. This is done by defining high level operations for *computation* and *communication* such as FIFO `read` and `produce`. Moreover, when it comes to share the execution resources within and between the applications, *scheduling* operations and the time at which the computation and communication operations execute play a significant role in executing the applications. Thus, a model of execution has to explicitly define computation, communication, and scheduling operations for each individual model of computation.

In this section, we first define computation and communication operations of our model of execution for the data-driven and time-driven models of computation, introduced in Section 2.5. Besides this, we propose the scheduling operations for each model of computation in accordance with the scheduling classification introduced in Section 2.4. Finally, in the context of the CompSOC platform, we present different variants of the model of execution in implementing a task's equivalence of a data-driven process and actor, and a time-driven job.

4.1.1 Execution Operations: Computation & Communication

The execution operations are categorized as: (i) computation, (ii) communication, and (iii) scheduling operations. In order to propose a unified model of execution that implements both data-driven and time-driven models of computation, we identify the common operation primitives required for executing these target models of computation. These common operations are presented in Table 4.1.

As presented in Table 2.1, when it comes to execution platforms, depending on an application's model of computation a task is equivalent to a KPN process, a CSDF actor, or a time-driven job. Having introduced the task temporal model in Section 2.4.2, the execution state of a task is therefore either ready, blocked, running, interrupted, or possibly finished. In this section, the execution operations of a task is described with respect to these task states.

In order to describe the essence of each execution operation in the different models of computation, we first abstract from the scheduling operations and focus on the computation and communication ones.

Table 4.1: *List of the execution operations required for executing the models of computation.*

Operation	Category	Description
C	computation	Performs the computational functions
F_C	communication	Checks if a (shared) data or space is present. If not, it blocks.
F_A	communication	Accesses a (shared) data or space
S_T	scheduling	Selects a next task to be scheduled as part of intra-application scheduling
W	Scheduling	Waits idle for an event or an interrupt to trigger the intra-application scheduler

A computation operation, C, is defined as the sequence of all instructions (except the instructions for communication) that implements a process/actor/job's functionality, and it is usually between two consecutive communication operations. This operation directly corresponds to the *compute* operation of the models of computation introduced in Section 2.5.

The communication between an application's tasks can be inter- or intra-processor-tile, depending on the mapping of the data producing and consuming tasks on the tiles. In a MPSoC platform with distributed shared memories, such as CompSOC, the difference between these two inter-tasks communication scenarios is in the actual memory locations that are accessed from a tile's processor. Our model of execution however abstracts from these two scenarios and proposes general communication operations for memory-mapped FIFO-based communication of data- and time-driven models of computation. In this method, the model of execution hides the details of FIFOs implementation and the actual memory-mapped location from the perspective of the applications executing on the processor. The model of execution ensures a transparent access to inter- and intra-tile memory locations, while the memory consistency is guaranteed by the communication model of the underlying platform [30].

A FIFO typically requires separate administration and access operations. A FIFO administration operation is to *check* for availability of space or data denoted with F_C. A FIFO buffer *access*, i.e., placing/retrieving data into/from the buffer, is represented as F_A. In data-driven models of computation such as KPN and CSDF, F_A may be performed only after making sure that space or data exist by an F_C operation. In the time-driven model of computation however F_A may be performed without any FIFO check required. As described in Section 2.5.2, the reason for this is that inter-jobs communication is realized by one-space FIFOs and a job assumes that the data and the space is always available.

In the rest of this section, the implementation of the models of computation by our model of execution, using these execution operations, are represented with a regular language sequence [42]. In this representation, if A and B are two operations, the regular expression language is defined as follows:

$$A^0 = \epsilon$$

$$A^{N+1} = AA^N, \text{ where } 0 \leq N < \infty$$

$$AB = \{ab | a \in A, b \in B\}$$

$$(A+B) = A \cup B$$

$$A^\# = A^{[0,N]} = \{\epsilon, A, AA, \dots, A^N\}, \text{ where } N < \infty$$

$$A^+ = A^{[1,\infty)} = \{A, AA, AAA, \dots\}$$

$$A^* = A^{[0,\infty)} = \{\epsilon, A, AA, AAA, \dots\}$$

Data-Driven Models of Computation

In what follows we present how the FIFO-based inter-task communications of KPN processes and CSDF actors are implemented with the aforementioned execution operations. A KPN process and a CSDF actor activation corresponds to a task firing in the model of execution. We define a task's status as *ready* if it can execute. If a task is not ready, its status is either *blocked* or *finished*.

A KPN process is initially ready by default, i.e., the body of the process starts executing, until it is either blocked on a FIFO read/write or it finishes. In other words, a KPN process (or its corresponding task) is activated once, and its body is implemented with a loop inside the process body over an interleaved sequence of computation and communication operations.

A KPN process may immediately start (it is ready), and it blocks whenever it executes a *read* or *write* for which there is not enough data and/or space. Once it has been activated, a KPN task is not guaranteed to finish without blocking, thus the task status is not fixed for its entire entire execution after an activation.

KPN *read* and *write* operations require both FIFO check and FIFO access. Formally, these operations are implemented as $F_C F_A$. Since F_C is blocking, this implies that a task possibly waits for data/space to becomes available for an unknown time and then can access it.

In the KPN model the *read* and *write* operations may be arbitrarily interleaved with *compute*, $(C + F_C F_A)^*$. Note that F_C represents a blocking check on a FIFO and $*$ models the possible infinite loop inside a process.

A CSDF actor is ready when it has enough data and space in the consuming and/or producing FIFOs, respectively. The task corresponding to an actor is activated when its firing rule is satisfied, and therefore, it can fire possibly for an infinite number of activations. For a CSDF actor, the corresponding task's status is unchanged for an entire iteration, namely ready.

The CSDF *produce* and *consume* operations require only FIFO access since the checking has been done by the firing rule before an actor becomes ready. Consequently, these operations are implemented as just F_A.

In CSDF the execution order is strict: it starts witch checking, i.e., F_C, all producing and consuming FIFOs according to the firing rules. Then the body begins with *consume* operations, i.e., F_A, continues with the *compute*, C, and ends with all *produce* operation, i.e., again F_A. Thus, the execution of a CSDF task can be represented by $(F_C^{\#} F_A^{\#} C F_A^{\#})$, where # models the number of FIFOs that has to be checked/accessed.

Such correspondence of each model of computation's operation with an execution operation is also presented in Listing 4.1 and Listing 4.2 beside an example pseudo code of a KPN process and a CSDF actor.

Time-Driven Models of Computation

In the case of the time-driven model of computation, when a job is ready to execute at an specific moment of time, it assumes that its required data and space are available. The model of execution therefore implements a job in a simple orchestration of interleaved one-place FIFO accesses and compute operations as $(C + F_A)^*$. Such correspondence of the model of computation's operation with an execution operation is also presented in Listing 4.3 beside an example pseudo code of a time-driven job.

```
    MoE     |
Operations  |                              KPN Process
------------|-------------------------------------------------------------------
            |/* Start of process body */
    $C$     |initialization();
            |
            |For (i=0; i<N; i++) {
  $F_CF_A$  |  x = read(<f4,1>); /* Read 1 data token from FIFO 4 */
            |
    $C$     |  P5_compute1(x);
            |
            |  if (x == 0) {
  $F_CF_A$  |    y = read(<f5,1>); /* Read 1 data token from FIFO 5 */
            |  } else {
    $C$     |    y = P5_compute2(x);
            |  }
    $C$     |  z = P5_compute3(y);
            |
  $F_CF_A$  |  write(<f6,1>, [z]); /* Write 1 data token to FIFO 6 */
            |}
            |
  $F_CF_A$  |write(<f6,1>, [a_final_token]); /* Write 1 data token to FIFO 6 */
            |
    $C$     |finalization();
            |/* End of process body */
```

Listing 4.1: *Execution operations corresponding to an example pseudo-code of a process in a KPN model of computation.*

```
    MoE     |
Operations  |                              CSDF Actor
------------|-------------------------------------------------------------------
            |/* Cyclo static firing rules */
            |switch cycle
            |  case 0:
   $F_C$    |    wait_until ( data_token_available_on_fifo(<f4, 1>, <f5, 1>)
            |                 AND
            |                 space_token_available_on_fifo( <f6, 2>) );
            |  case 1:
   $F_C$    |    wait_until ( data_token_available_on_fifo(<f4, 1>, <f5, 0>)
            |                 AND
            |                 space_token_available_on_fifo( <f6, 1>) );
            |End
            |
            |/* Start of actor body */
   $F_A$    |x = consume(<f4,1>);  /* Consume 1 token from FIFO 4 */
            |
   $F_A$    |IF (cycle == 0) y = consume(<f5,1>); /* Consume 1 token from FIFO 5 */
    $C$     | w = P5_compute1(x);
    $C$     |IF (cycle == 0) z = P5_compute2(y);
            |
            |IF (cycle == 0) {
   $F_A$    |  produce(<f6,2>, [w,z]); /* Produce 2 tokens to FIFO 6 */
            |} ELSE {
   $F_A$    |  produce(<f6,1>, [w]);   /* Produce 1 token to FIFO 6 */
            |}
            |
            |/* End of actor body */
            |cycle = (++cycle) MOD 2; /* Increment cycle for the next activation */
```

Listing 4.2: *Execution operations corresponding to an example pseudo-code of an actor in a CSDF model of computation.*

```
   MoE      |
Operations  |                            Tim-Driven Job
------------|---------------------------------------------------------------------
            |/* Start of job body */
    F_A     |x = read(<f0,1>);
    C       |y = J7_compute(x);
    F_A     |write(<f1,1>, [y]);
            |/* End of job body */
```

Listing 4.3: *Execution operations corresponding to an example pseudo-code of an actor in a time-driven model of computation.*

4.1.2 Execution Operations: Scheduling

When multiple applications execute on a single platform, conventionally, the scheduling is done at two hierarchical levels of inter- and intra-application. In CompSOC platform, since every application is executing in a dedicated virtual platform which corresponds to a partition on every processor tile, the inter-application scheduling is handled by the CoMik kernel, as explained in Chapter 3. However, when it comes to executing an individual application implemented by a model of computation, the intra-application scheduling operations have to be realized by the model of execution.

The intra-application scheduler, known also as the task scheduler, determines the next task of the application to execute. In general, the model of execution defines S_T as the execution operation that selects a task according to a given policy, e.g., TDM and Round-Robin (RR). As described in the scheduling classification of Section 2.4, a task scheduling policy may be either cooperative/preemptive time- or event-triggered. In general, if a ready task is not found, an *idle* task is scheduled. The idle task does nothing other than waiting until the next invocation of the task scheduler.

For each model of computation, the scheduling operations are composed and orchestrated by considering the properties of the model. In KPN any policy can be used to schedule a task although scheduling a ready task is more reasonable and efficient. Thus scheduling a KPN task can be implemented as just an S_T operation. In CSDF the task scheduler has to first find a ready task. For this, it selects a task, S_T, and then, it checks each of its FIFOs, $F_C^{\#}$, repeatedly. This sequence formally results in $(S_T F_C^{\#})^+$.

In the time-driven model of computation, when the scheduler is invoked all the jobs (tasks) are already ready. Therefore, the next task can be simply selected based on a policy (for example, select the task that has earliest deadline) by an S_T operation among all the ready tasks.

The model of execution does not care about the triggering mechanism of the scheduler. However, whether the scheduler is preemptive or cooperative, the execution operations of a task may differ. In case of cooperative scheduler, the executing task is allowed to finish and then the scheduler is invoked. Whereas, in case of preemptive scheduler, the executing task is either gets preempted by an event or when it has finished its entire execution it has to wait for an event. In the latter case, an extra waiting operation, denoted as W, is defined by the model of execution. This operation implements a waiting time for an event or an interrupt that triggers the scheduler. For example, when a KPN task's body, $(C + F_C F_A)^*$, executes and it has to wait for an interrupt to occur, the model of execution is realized like this: $(C+F_C F_A)^* W$.

The following section presents how the model of execution realizes the models of computation with the operations that have been introduced here.

4.1.3 Realization of Models of Computation

Table 4.2 presents the model of execution that realizes KPN, dataflow (specifically, CSDF), and time-driven models of computation. For the sake of clarity we demonstrate the representation of the model of execution for each model of computation in the case of cooperative and preemptive scheduling separately, although it is possible to come up with a more complex compact version of the model of execution.

Moreover, the underlined operations in the table indicate the body of tasks, i.e., KPN processes, CSDF actors, and time-driven jobs.

Table 4.2: *Implementation of the models of computation with the unified model of execution when task scheduling is either cooperative or preemptive.*

Model of Computation	Model of Execution	
	Cooperative	Preemptive
KPN	$(S_T\underline{(C+F_C F_A)^*})^*$	$(S_T\underline{(C+F_C F_A+F_A)^*}W)^*$
CSDF (dataflow)	$(S_T F^\#_C(\underline{F^\#_A C F^\#_A}))^*$	$(S_T(F^\#_C\underline{F^\#_A C F^\#_A + F^\#_A C F^\#_A + C F^\#_A + F^\#_A})^+ W)^*$
Time-Driven	$(S_T\underline{(C+F_A)}^*)^*$	$(S_T\underline{((C+F_A)W)^*})^*$

For KPN, in case of cooperative scheduling, the model of execution is simply implemented by composing a number of interleaved computation and communication operations immediately after a scheduling operation. A task selection has to be repeated whenever a task has finished. In details, after a task is initially selected, i.e., S_T in Table 4.2, 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, basically the current task is blocked, thus instead of polling for data or space, the task may yield the execution to the task scheduler (i.e, cooperatively invoke the scheduler) so that another task is selected. In such scenario, the execution continues from executing another S_T. Otherwise, the FIFO 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. when a KPN task finishes, it would not be scheduled anymore.

For KPN, in case of preemptive scheduling, the normal execution is exactly the same as the cooperative one, but, the task can be possibly preempted in the middle of C or F_A operations. In such cases, the execution continues by a scheduling operation and a new scheduled task's operation. When the preempted task is scheduled back, its execution resumes from the operation that has been preempted, which is either C or F_A operation. To cover this case, the model of execution, in Table 4.2, includes an individual F_A in the body of the process. Furthermore, when a task finishes, the task has to wait for an interrupt. This is modeled by the W in Table 4.2.

The model of execution for CSDF, in case of cooperative scheduling, is basically a composition of $F_A^{\#}CF_A^{\#}$, as the task body, and $S_T F_C^{\#}$, as the task scheduling. Since a CSDF actor activates for as many times as its firing rules are satisfied, the task body in this model possibly repeats for an infinite number of times. Thus, a CSDF task never finishes. But, at the end of each activation of a task, the task scheduler is called immediately to re-decide about the task that has to execute next.

For CSDF, in case of preemptive scheduling, the model of execution is similar to the case of preemptive scheduling for KPN. The CSDF task's body may be preempted in the middle of C or F_A operations, and when it comes back to resume, the possible execution operations are distinguished as: (i) $F_A^{\#}CF_A^{\#}$: when it has been preempted in consume operation which is the first FIFO access of the body, (ii) $CF_A^{\#}$: when it has been preempted in the compute operation of the body, and (iii) $F_A^{\#}$: when it has been preempted in the produce operation which is the final FIFO access of the body. Furthermore, when a task finishes, in this case, it has to wait for an interrupt to occur. This is modeled by the W in Table 4.2.

For the time-driven model of computation, the model of execution implements each job's body with a sequence of interleaved one-place FIFO access (without any need to check for data or space) and computation operation as $(C+F_A)^*$, where each task (job) starts with an S_T operation. In case of preemptive scheduling, the model of execution is presented the same as the cooperative one, although the tasks may possibly preempted int the middle of C or F_A operations. Besides this, in preemptive scheduling, the task finishes with a waiting operation for an interrupt.

4.1.4 Discussion on Realizing Models of Computation with the Model of Execution

Having presented how the model of execution implements each model of computation in Table 4.2, in this section, we elaborate on predictability of the model of execution. Furthermore, we discuss the performance trade-offs between the cooperative and preemptive scheduling variants of the model of execution for each model of computation.

Predictability

As one of the dataflow variants, CSDF model of computation is analyzable, since the status of its task is *constant during an activation* after the firing rules have been checked. This constant status is visible in the realization of the dataflow with the model of execution in Table 4.2, where no F_C operation exists in a CSDF actor/task's body to cause unpredictable blocking time on a communication [2]. The operations in the CSDF body execute for a bounded number of repetitions (including only #, and no $*$ or $+$). Assuming that each operation finishes in a bounded time (as described in Chapter 2, it is a valid assumption on the CompSOC platform), the model of execution corresponding to the CSDF model is predictable and therefore amenable to temporal analysis. Consequently, the CSDF model suits FRT applications as well as soft and NRT ones.

[2] A task's body is distinguished with the underlined operations.

In KPN, as presented in Table 4.2, since F_C executes before a FIFO access in the body, the task may be blocked. Moreover, a possibly infinite loop in the body, that is modeled by $*$, results in an unknown execution time of the task. These make the exact timing analysis of the KPN impossible. Thus, KPN does not suit FRT applications, but only SRT and NRT applications. Moreover, KPN can model dynamic behavior of applications, since the order and orchestration of *read*s and *write*s and the number of tokens that can be accessed are arbitrary. As a result, the status of a KPN task is not available before giving the control to that task, and leads to less possible scheduling optimizations.

Scheduling Trade-offs

In case of preemptive scheduling of the models presented in Table 4.2, when a KPN process, a CSDF actor or a time-driven job has been finished, a waiting operation, W, executes until the moment that an event/interrupt occurs. This time is therefore wasted and this approach is *non-work-conserving*, potentially leading to a low processor utilization. To prevent such an undesirable situation, the waiting time has to be made as small as possible, preferably zero, by aligning the occurrence of events or interrupts with the exact moment that a task finishes (i.e., the time that it produces output data tokens, that can make other tasks become ready). The intrinsic behavior of the time-driven models of computation allows such a practice as the start and finish time of each job is a-priori known. Thus, the preemptive scheduling well suits this model.

On the other hand, the blocking times that may occur in a KPN process body are not predictable and therefore it is not really possible to shrink/remove the waiting times. Subsequently, as a *work-conserving* method, the cooperative scheduling is the best fit for executing KPN model of computation when a worst-case execution time analysis can be done [67, 92]. Even when a KPN task is predictably blocked on a FIFO check, it can cooperatively yield the control so that the task scheduler is invoked. However, when the behavior of a KPN task is not predictable, in order to avoid deadlock, preemptive scheduling is preferred.

In case of CSDF, although there exist some research efforts to align the finishing time of actors with preemptive scheduling interrupts [11], the intrinsic behavior of the CSDF models (as explained in Section 2.5) is well compatible with cooperative schedulers that a new task may be scheduled immediately after a finished task.

4.2 CompOSe: an Operating System Library

The model of execution is implemented on the CompSOC platform in a form of an OS library, namely CompOSe. As illustrated in Figure 4.2, the CompOSe library executes in a *partition* created by the CoMik microkernel to execute an application. CompOSe implements the execution primitives proposed by the model of execution. The application uses the API provided by CompOSe in order to execute its *computation*, *communication*, and *scheduling* operations according to its model of computation. In principle, CompOSe is an untrusted code that runs in partition-mode while its access to the underlying platform resources are controlled and restricted by the microkernel.

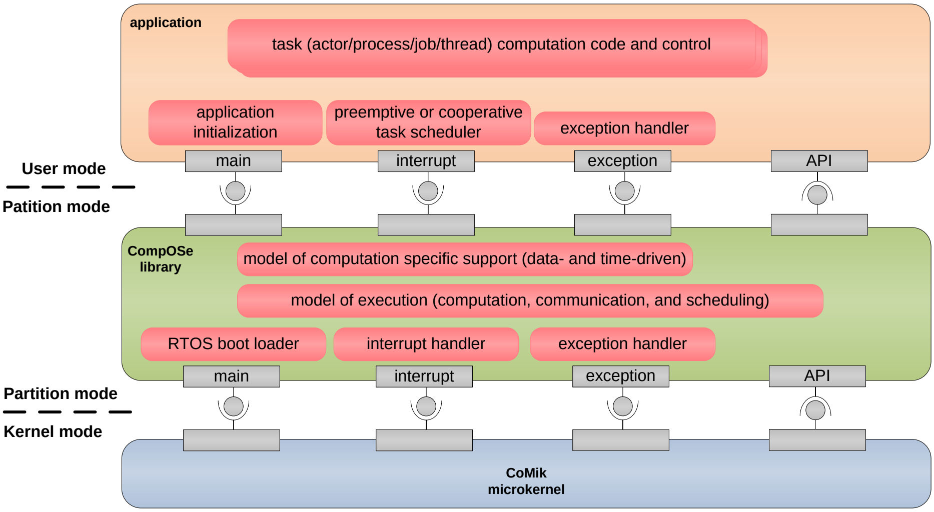

application
task (actor/process/job/thread) computation code and control
application initialization
preemptive or cooperative task scheduler
exception handler
main
interrupt
exception
API
User mode
Patition mode
CompOSe library
model of computation specific support (data- and time-driven)
model of execution (computation, communication, and scheduling)
RTOS boot loader
interrupt handler
exception handler
main
interrupt
exception
API
Partition mode
Kernel mode
CoMik microkernel

Figure 4.2: *The structure of the model of execution implemented in form of CompOSe OS library.*

In the context of the predictable system, the CompOSe is implemented in such a way that it does not introduce any uncertainty in executing an application on top of the CoMik microkernel. CompOSe can be seen as a simple RTOS since it schedules tasks of an application on the basis of both data and time events.

The detailed structure of the CompOSe library is presented in Figure 4.2, where its implementation is clearly divided into the software units for: (i) providing the *main*, *interrupt handler*, and *exception* handler software hooks required by CoMik from a partition, (ii) implementing the execution primitives of the model of execution using the API provided by CoMik, and (iii) giving model of computation specific support to the application by providing Software Containers (SCs).

In the rest of this section, we are doing to describe detailed structure of the CompOSe OS library, and we demonstrate how different models of computation can be supported in the implementation of the model of execution.

4.2.1 CompOSe Data Structure

The CompOSe library is created in the form of an Application Control Block (ACB) and linked to its corresponding PCB. As described in Section 3.3.3, every partition has its own dedicated *heap* memory section. The ACB is created at the beginning of the partition's heap, and the pointer to the parent partition is kept in the ACB for further partition's resources access. Figure 4.3 illustrates an abstract overview of an ACB generic structure.

Every application, expressed in either of KPN, CSDF, or time-driven models of computation, consists of a set of tasks. The tasks are created with a unique identification in a linked list of Task Control Blocks (TCBs). Besides this, the model of computation that the application is expressed with is stored in the ACB for performing the model of computation specific operations.

Possibly, every application uses its own task scheduler, and therefore, the scheduler has to be provided to CompOSe. Depending on the given scheduling strategy, i.e., cooperative or preemptive, CompOSe schedules the tasks of the application accordingly, and sets the current running task.

Furthermore, the execution timing information of the application is updated and kept for performance monitoring in a timing control block.

A task control block is generic enough to support KPN processes, CSDF actors, and time-driven jobs. Each of these different tasks needs a computation function, a dedicated stack memory, state, and timing control block. The reason for a dedicated stack memory is that in case of preemptive task scheduling, every task may get interrupted in the middle of its execution and later it has to be resumed exactly from the same point. Thus, its execution status has to be stored and recovered from its own dedicated stack.

Depending on the model of computation, the computation control block of the tasks is different for each application. This is due to the fact that the application's task in every model of computation has different computation properties. For example, a CSDF task needs to be provided with a firing rule updating function so that CompOSe can check its firing rules before each activation, according to the model of execution. As another example, in case of preemptive scheduling, a time-driven task needs to be provided with the task properties such as release time, inter-release time, deadline and workload (introduced in Section 2.4.2).

In theory, every application may use its own communication mechanism between its tasks. In practice, in the CompSOC platform, the all our target models of computation uses FIFO-based communications. Thus, for each task, the communication control block includes a set of FIFO Control Blocks (FCBs) each of which is created and instantiated for a FIFO in the model of computation. Each FCB consists of all the necessary data elements for FIFO administration, such as the pointers to its consuming and producing tasks, buffer size, data and space pointers, read and write counters, etc..

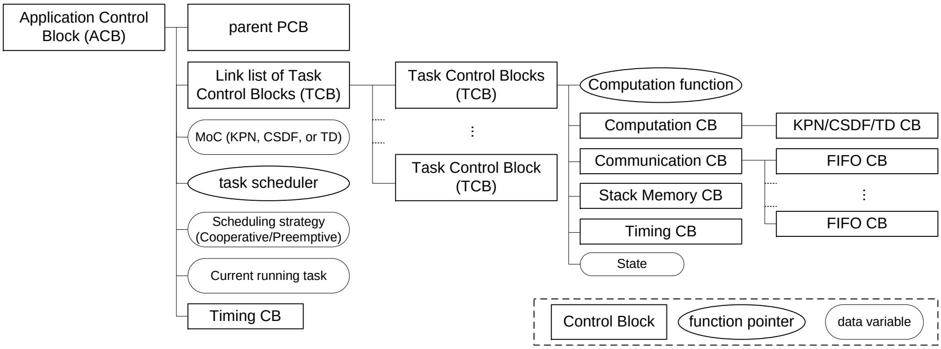


Figure 4.3: *Data structure of CompOSe*

After initializing the application, CompOSe uses these control blocks to schedule and control each application's task. For this purpose, CompOSe has to, (i) provide the software hooks of main, interrupt and exception handler to CoMik in order to attach the library to the virtual platform, (ii) implement the model of execution by realizing the primitive execution operations in form of an API, and (iii) provide the a set of SCs so

```
void OS_main() {

    /* Allocate memory for the ACB */
    os_initialise_libcompose();

    /* Creating and initializing the task scheduler arguments */
    ...

    /* Add an application to the current partition */
    os_add_application(Application_ID, NUM_TASKS, TASK_SCHEDULER);

    /* Set the application's model of computation */
    os_set_application_moc([KPN, CSDF, or time-driven]);

    /* Set the task scheduling strategy */
    os_set_task_scheduling_strategy([COOPERATIVE_TASK_SCHEDULING or
         PREEMPTIVE_TASK_SCHEDULING]);

    /* Add tasks with their corresponding IDs */
    os_add_task(TASK1_ID, TASK1_STACK_SIZE, TASK1_COMPUTATION_FUNCTION);
    os_add_task(TASK2_ID, TASK2_STACK_SIZE, TASK2_COMPUTATION_FUNCTION);
    ...
    os_add_task(TASKn_ID, TASKn_STACK_SIZE, TASKn_COMPUTATION_FUNCTION);

    /* Set the computation control block */
    ...

    /* Set the communication control block */
    ...

    /* Start the application */
    os_application_start();
}
```

Listing 4.4: *Pseudo code representation of CompOSe boot-loader.*

that the various applications expressed in the different models of computation can be executed. In the rest of this section we explain each of these points, in turn.

4.2.2 Partition Software Hooks

CompOSe uses the *main*, the *interrupt handler*, and the *exception* handler software hooks offered by CoMik, as illustrated in Figure 4.2.

The boot-loader of CompOSe is the main function that a partition starts executing after it is created by CoMik. In this function, the CompOSe library is initially created in the form of an ACB and linked to its corresponding PCB. Afterward, the control is immediately given to the application initialization procedure. Listing 4.4 presents a pseudo code of the main function. Using the underlying API provided by CompOSe, this function properly populates the ACB according to the application's model of computation.

The CompOSe interrupt handler is required to be provided to CoMik only in the case that the application uses preemptive (time-triggered) scheduling. In this case, the interrupt handler hooks the preemptive task scheduler to the CoMik scheduler as an *integrated partition interrupt handler* as described in Section 3.6. For this purpose, the CompOSe interrupt handler, as presented in Listing 4.5, simply calls the task scheduler.

```
void os_interrupt_handler() {

  /* Acknowledge the partition interrupt */
  os_ack_current_partition_interrupt();

  /* Call the task scheduler registered to the ACB */
  os_schedule_task();

  /* Enable the partition interrupt for the next interrupt */
  os_enable_current_partition_interrupt();

}
```

Listing 4.5: *Pseudo code representation of CompOSe interrupt handler.*

The task scheduler itself is provided by the user, and it runs in user-mode.

Regarding the exception handling, currently all the exceptions are handled by CoMik and there is no need for CompOSe to provide its own exception handler.

4.2.3 Implementing the Model of Execution

CompOSe runs in partition mode meaning that it is only allowed to use the API provided by CoMik to access the underlying platform resources that belong to its own partition. In turn, CompOSe provides a high-level API to the applications running in *user-mode*. In this mode, the applications are only allowed to use the API that realizes the model of execution corresponding to their model of computation. This consists of the API for communication and scheduling operations, whereas the computation operation does not need any special attention.

Communication API

For the FIFO-based communication operations, CompOSe follows the C-HEAP protocol [78]. To implement this protocol, the following functions are realized by CompOSe:

- `claim_fifo_data_token()` to acquire a data token for data consumption.
- `claim_fifo_space_token()` to acquire a space token for data production.
- `release_fifo_data_token()` to release a produced token.
- `release_fifo_space_token()` to release a consumed token.

The number of tokens that can be claimed/released from a FIFO at a time is defined as the FIFO rate. Using the above APIs, one token can be accessed at a time. For accessing more than one token the above APIs has to be executed in a loop. The following API is proposed to set the consumption and production rate of a FIFO:

- `os_set_fifo_consumption_rate(fifo_id,rate)` to set the consumption rate of the FIFO.

- `os_set_fifo_production_rate(fifo_id,rate)` to set the production rate of the FIFO.

In case of claiming a data/space token, if the aforementioned functions returns with a *NULL* pointer, it means that the token is not available. This result can be used to implement FIFO check operations as follows:

- `os_check_fifo_data_token(fifo_id,rate)` to check if a given number of data tokens are available in the FIFO.
- `os_check_fifo_space_token(fifo_id,rate)` to check if a given number of space tokens are available the FIFO.

Furthermore, the common FIFO access operations proposed by the model of execution for all the models of computation, is to set or get data tokens. Using the above functions, CompOSe provides the following API for high-level FIFO access operations:

- `os_get_fifo_data(fifo_id,rate,*data_in)` to get a number of data tokens from the FIFO.
- `os_set_fifo_data(fifo_id,rate,*data_out)` to set a number of data tokens into a FIFO.

In case of the KPN and the time-driven models of computation, the FIFO read and write operations are implemented using the claim and release API before these get and set API. As a result, the following high-level API is available to developers:

- `os_read_fifo(fifo_id, rate,&data_in)` to check the FIFO for the available data tokens and receives them.
- `os_write_fifo(fifo_id, rate,&data_out)` to check the FIFO for the available space tokens and sends them.

In case of CSDF, as the FIFO checks happens in scheduling time, the following API is provided to developers to access the data and space tokens from a task's body:

- `os_get_csdf_task_data_in_ptrs()` to get pointers to the data tokens.
- `os_get_csdf_task_data_out_ptrs()` to get pointers to the space tokens.

Using these high-level API all the communication operations of the model of execution can be implemented for the models of computation.

Scheduling API

For every application, CompOSe is provided with a task scheduler. Although every task scheduler may use different policy, there are a number of common facilities needed by them in order to perform the scheduling operations. The common facilities can be distinguished as a set of API to access each task's computation and communication control blocks in its TCB, as well as the restricted API to access the corresponding ACB. Since the first API is different per model of computation, here, we give two examples of such API for CSDF and time-driven models of computation.

In case of CSDF, CompOSe enables the application's scheduler to check the firing rules of a task using the following API:

- `os_check_firing_rules(task_id)` to check the firing rules of a task with ID of *task_id* in the application.

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

A firing rule is the number of tokens (or rate) that has to be consumed/produced in an activation of a CSDF task in its associated consuming/producing tasks. In order to enable a CSDF modeled application to set its tasks firing rule, the application has to provide a *firing rule updating* function for every task. In this function, the application can set each FIFO's rate in every activation according the cycle of the task. For this, CompOSe provides the following API:

- `os_get_csdf_task_cycle()` to return the cycle that the task currently execute in.
- `os_get_cons_fifo_rate_table()` to return the rate table of all consuming FIFOs associated to the tasks. Using this table, the consumption rates of all the consuming FIFOs is set for the next activation.
- `os_get_prod_fifo_rate_table()` to return the rate table of all producing FIFOs associated to the tasks. Using this table, the production rates of all the producing FIFOs is set for the next activation.

To exemplify the use of this API, assume that a CSDF task has the firing rule of $fr = \{(<1,3>, <3,2>), (<1,2>, <2,1>)\}$, which means the task has two cycles: (1) FIFO 1 has to be checked for 3 tokens and FIFO 3 has to be checked for 2 tokens, and (2) FIFO 1 has to be checked for 2 tokens and FIFO 2 has to be checked for 1 token. Here, we ignore the details of which FIFO is producing or consuming. Depending on the cycle that task is in, the above API has to be called by the firing-rule updating function to set the rate of FIFOs 1 and 3 to 3 and 2, respectively, in the first cycle, and to set the rate of FIFOs 1 and 2 to 2 and 1, respectively, in the second cycle.

In case a time-driven application, CompOSe enables the application's scheduler to get for example a task's properties such as work-load, priority, deadline, release time, etc. Based on these information, a task may then gets scheduled.

The common facilities that are required by all kinds of scheduler in every model of computation is provided by CompOSe in a set of the following API:

- `os_get_task_state(task_id)` return a task's state which can be either interrupted or ready.
- `os_get_current_task()` return the tasks that has been running before the scheduler is invoked either preemptively or cooperatively.
- `os_set_next_task(task_id)` set the next task to be scheduled by CompOSe.

Using these scheduling API, an application can perform all the necessary operations introduced by our model of execution in order to schedule the tasks in either of KPN, CSDF, or time-driven model of computation.

4.2.4 Software Containers

CompOSe proposes a set of SCs for developers to implement applications expressed in one of the KPN, CSDF, and time-driven models of computation. These software containers are realized to match the model of execution corresponding to each model of computation. Here, we describe each of these containers to implement tasks' computation, communication, and scheduling operations.

Computation & Communication

The computation SCs includes a task's body and some task's model of computation-specific containers. The body container is a generic `void` function with no specific argument needed to implement all the different tasks of the models of computation. Listing 4.6 presents an example implementation of the KPN task, introduced in Listing 4.1, using the body container and the computation and communication API.

The implementation of the time-driven task is very similar to the KPN's one. In case of CSDF, however, Listing 4.7 presents an example implementation of the actor, introduced in Listing 4.2, using the body container and the computation and communication API.

An example of the model of computation-specific containers is the one needed for implementing the firing rules of CSDF actors. Listing 4.8 presents an instance of such container that implements a set of firing rules using the API introduced earlier in this section. In this example, the task may fire if it has one data and one space tokens in its first consuming and producing FIFO.

Scheduling

The SC of task scheduler has to support both preemptive and cooperative schedulers with different scheduling policies. For this purpose, the container is a generic `void` function with no specific argument. Any specific data structure needed for scheduling can be held in the computation and communication control blocks per task.

Using the scheduling API introduced earlier in this section, different types of task scheduler can be then implemented in the container. An example of such container is

```
void task_body_container () {

  int N;
  int i;
  int x,y,z;

  /* Initialization */
  int N = 100;

  For ( i=0; i<N; i++) {

    /* Read one token from FIFO 4 */
    os_read_fifo(FIFO_ID_4, 1, &x);

    /* Do some computation */
    x += 10;

    if ( x==0) {
    /* Read one token from FIFO 5 */
      os_read_fifo(FIFO_ID_5, 1, &y);
    } else {
      y = x * 10;
    }

    z = y ^ 2;

    /* Read one token from FIFO 6 */
    os_write_fifo(FIFO_ID_6, 1, &z);
  }

  /* Finalization */
  z = N;
  os_write_fifo(FIFO_ID_6, 1, &z);
}
```

Listing 4.6: *Pseudo code representation of task's body container for KPN processes.*

```
void task_body_container () {

  int* data_in, data_out;
  int x, y;
  int w, z;

  /* Get the current CSDF cycle */
  int csdf_cycle = os_get_csdf_task_cycle();

  /* Get the tokens */
  data_in = (int*) os_get_csdf_task_data_in_ptrs();
  data_out = (int*) os_get_csdf_task_data_out_ptrs();

  x = data_in[0];

  if ( csdf_cycle == 1) y = data_in[1];

  /* Do some computation */
  w = x * 10;

  if ( csdf_cycle == 1) z = y ^ 2;

  /* Write the produced data into producing tokens */
  if ( csdf_cycle == 1) {
    data_out[0] = w;
    data_out[1] = z;
  } else {
    data_out[0] = w;
  }

}
```

Listing 4.7: *Pseudo code representation of the task's body container for CSDF actors.*

```
void firing_rule_container () {

  int* cons_fifos = os_get_cons_fifo_rate_table();
  int* prod_fifos = os_get_prod_fifo_rate_table();

  int csdf_cycle = os_get_csdf_task_cycle();

  Switch cycle
    case 1:
        cons_fifos[0]=1;
        cons_fifos[1]=1;

        prod_fifos[0]=2;

    case 2:
        cons_fifos[0]=1;
        cons_fifos[1]=0;

        prod_fifos[0]=1;
  End
}
```

Listing 4.8: *Pseudo code representation of the firing-rules container for CSDF actors.*

```
void task_scheduler_container (){

  bool can_fire = 0;
  int current_task_id;
  int next_task_id;

  current_task_id = os_get_current_task();

  if ( current_task_id == 1) {

    selected_task_id = 2;
    can_fire = os_check_task_fr(next_task_id);
  } else {
    selected_task_id = 1;
    can_fire = os_check_task_fr(next_task_id);
  }

  if ( can_fire) {
    /* Schedule the next task */
    os_set_next_task(next_task_id);
  } else {
    /* Schedule an IDLE task */
    os_set_next_task(0);
  }
}
```

Listing 4.9: *Pseudo code representation of a task scheduler container for a CSDF actor.*

presented in Listing 4.9 for a CSDF task. In this example, the task scheduler switches alternatively between two tasks. It checks the firing rule for the tasks, and if the selected task cannot fire, it schedules the IDLE task.

In summary, using these software programming containers, CompOSe enables developers to implement any application expressed in KPN, CSDF, or time-driven models of computation. Consequently, a unified support for different models of computation is provided by the model of execution.

4.3 Related Work

In the context of proposing and realizing a model of execution using a RTOS, existing work falls into three categories: (i) mapping and implementation of the various models of computation, specifically the KPN and dataflow, (ii) 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 models of execution 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 [14, 16, 55]. 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 [8, 16, 17, 25, 39, 40, 90, 104]. Except the work presented in [16], none of them targets MPSoC platforms. The authors of [16] 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 [17] and [8] differ from ours in the sense that in [17] KPN processes are scheduled and executed on hardware reconfigurable accelerators, and in [8] not multiple applications may execute concurrently on the platform.

In case of dataflow models, applications performance can be accurately analyzed using several dataflow models [12, 79, 99]. Thus dataflow is used to express real-time applications executed on MPSoCs [55, 68]. 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 [55] 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 (weak) composability is very similar to the ones in [16] and [68], and, as mentioned above our definition, i.e., strong composability, 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. ForSyDe [84] 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 [105] and Giotto [36] 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 time-driven models [61]. It only supports applications with FRT 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 [20]. 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 support only time-driven applications.

Moreover, CASSE [82] 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 [45] 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 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- and time-driven models are implemented easily on the actual MPSoC platform.

4.4 Summary

In this chapter, we have introduced the requirements of the model of execution and how such a model is realized on CompSOC's virtual platforms. We have presented the implementation of the model of execution by the CompOSe library, and how different models of computation can be developed to execute on the platform.

CHAPTER 5

Case Studies

THE purpose of the composable virtual platform, described in this dissertation, is to execute mixed criticality applications realized with either data-driven or time-driven models of computation. In order to demonstrate empirically how the platform achieves this objective, we have to study the specific features of the platform as (i) predictability, (ii) composability, and (iii) the model of execution that implements multiple models of computation.

In this chapter, case studies are performed on two experiment setups: (i) Matlab simulation of a virtual platform, specifically simulating temporal behavior of CoMik in partitioning the processor tiles, to investigate the predictability of time-driven applications executing on the platform, and (ii) actual execution of real application use-cases on an FPGA prototype of the platform, to investigate the composability and multiple models of computation. Here, we skip the predictability experiments for data-driven application as the dataflow analysis for the similar systems has already been performed completely in the literature [29, 91], and explicitly for CompSOC platform when using CoMik in this thesis [75].

The rest of this chapter, therefore, continues with focusing on the experiment results of the two setups. In each section, we first introduce the details of each setup, following with explaining the use-cases that are going to be used in each study. The experiment results are then investigated with respect to predictability, composability, and support for multiple models of computation.

5.1 Predictability of Time-Driven Applications

A system is predictable if it is possible to define the timing behavior of the applications at design time. For this, timing analysis of the system is performed. Timing analysis of a real-time system is based on response time (*rsp* as defined in Section 2.4.2) of running applications. The *response time* of an application can be computed using a model methodology from response time of its tasks. Response time of a task is defined as the duration between the task's release time and the time that it finishes its execution, i.e. the output data/result is released in the corresponding producing FIFOs. The task τ_i is schedulable on a system, if and only if $rsp_i \leq d_i$ in all situations. This condition states that the response time of a task should be always less than or equal to its relative deadline. When a processor is shared by multiple tasks, the logical execution time (as defined in Section 2.4.2 of Chapter 2) and ultimately the response time of a task may increase if there is any higher priority task that is ready to execute. In the case that the processor is further shared between multiple applications each running in an isolated partition, such as in the CompSOC platform, the response time of all tasks of an application increases because they are blocked for the slots of other partitions. Besides this, the relative deadline of the tasks may be logically moved earlier because the exact moment of a deadline is in the blocking time of the application and therefore the task should finish before the end of the last allocated slot to the application's partition so that it meets the deadline. As a result, in such systems, the response time analysis takes also the allocation of system slots to applications (i.e., partitions' parameters) into account, and the condition of $rsp_i \leq d_i$ becomes tighter.

A number of different approaches that address these problems exist in the literature. Various analysis methods for so called temporal-partitioning systems have been proposed [7, 19, 65, 87]. These methods are not directly applicable to our platform, since contrary to all the existing systems, in our case, composability implies a fixed system slot overhead before every application slot, as illustrated in Figure 3.3. Thus, a response time analysis of applications running on the CompSOC platform is necessary in order to enable application developers following two design options:

1. (re-)design (legacy) applications to be schedulable on the CompSOC platform.

2. adjust the partition parameters, i.e., slot allocations and slot sizes, so that the available (legacy) applications are schedulable.

We first formalize the timing properties of the composable platform. As an example, a TDM inter-application schedule with the system period of T time-units and $M = 6$ number of slots is presented in Figure 5.1. The size of each slot, which consists of a partition slot, S_p, plus the system slot, S_s, is $S = \frac{T}{M}$ time-units. Application j is allocated a partition with a set of $m_j = 3$ distributed slots (blue blocks) in the system period.

The cumulative processor time that is exclusively available for the execution of application j until a time moment, t, depends directly on the size of its partition. This cumulative time corresponds to sum of all the allocated slots to the application j until time t, e.g., the blue slots in our example. The function that calculates this cumulative time is called *server characteristic* function in [59], *server supply* function in [19], *resource supply bound* function in [88], and *availability* function in [7]. Here, we also use the term of availability function and define it for an application j as follows.

$$A_j(t) = \left\lfloor \frac{t}{T} \right\rfloor \times A_j(T) + A_j(\Delta t) \tag{5.1}$$

where, $A_j(T) = m_j \times S_p$ is the cumulative available time for the application j in a system period and, Δt is the time duration left until t in the last system period as depicted

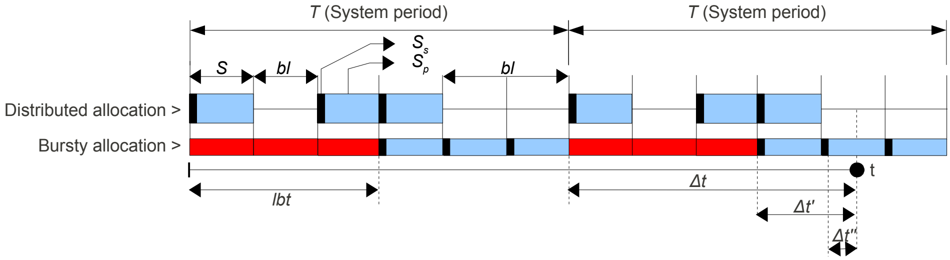


Figure 5.1: *An example of slots allocation to a partition (application) in a temporally-partitioned system, illustrating the cumulative available processing time and the (longest) blocking time of the partition.*

in Figure 5.1, and it is calculated as follows:

$$\Delta t = \begin{cases} t - T \times \lfloor \frac{t}{T} \rfloor & t < T \\ t & t \geq T \end{cases} \tag{5.2}$$

$A_j(\Delta t)$ depends on how the system slots are allocated to the application j. In other words, it depends on when the application is blocked by the execution of other applications. According to an existing theorem [65], when the processor allocation to an application is distributed, schedulability should be tested for every task at all the critical instances, i.e., the time when all the tasks are released at the beginning of a blocking time. Here, we propose a *conservative* approach which considers the longest blocking time (*lbt*) instance of an allocation. This is the case when all the slots that are not allocated to the application are consecutive at the beginning of the system period, as illustrated in Figure 5.1. In other words, the longest blocking time instance is when the allocation of slots to an application is bursty. Given such an allocation, the longest blocking time for an application j in a period is calculated with the following equation.

$$lbt_j = T \times \left(1 - \frac{m_j}{M}\right) \tag{5.3}$$

Therefore, the lower bound on $A_j(\Delta t)$ is defined as $A_j^l(\Delta t)$ and it is calculated as follows.

$$A_j^l(\Delta t) = \begin{cases} A_j^l(\Delta t') & \Delta t > lbt_j \\ 0 & \text{otherwise} \end{cases} \tag{5.4}$$

where, $\Delta t' = (\Delta t - lbt_j)$, and $A_j^l(\Delta t')$ is calculated as follows.

$$\Delta t'' = \left(\Delta t' - S \times \left\lfloor \frac{\Delta t'}{S} \right\rfloor\right), \text{and } S_p' = \left\lfloor \frac{\Delta t'}{S} \right\rfloor \times S_p$$

$$A_j^l(\Delta t') = \begin{cases} S_p' & \Delta t'' \leq S_s \\ S_p' + (\Delta t'' - S_s) & \Delta t'' > S_s \end{cases} \tag{5.5}$$

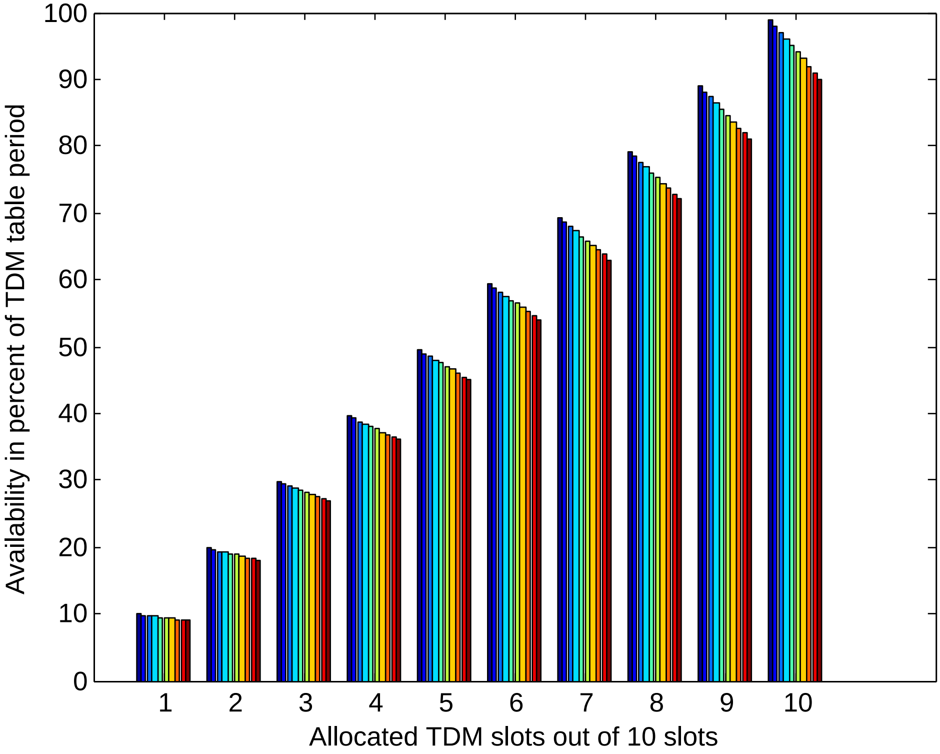


Figure 5.2: *Availability function of a partition in a 10-slot ($M = 10$) system when the slot allocation (m) is changing from 1 to 10. In each case, CoMik slot overhead (S_s) is varied from 1 to 10 percent of a partition slot, resulting in the different color bars.*

Finally, using the longest blocking time of an application, a lower bound on the availability function is given with the following equation.

$$A_j^l(t) = \left\lfloor \frac{t}{T} \right\rfloor \times A_j(T) + A_j^l(\Delta t) \tag{5.6}$$

Every application has its own constraints over the minimum required availability of the processor in a system's TDM period in order to meet its timing and functional requirements. Figure 5.2 illustrates the availability function of a partition in a 10-slot system, when the size of the partition, i.e., the number of slots allocated to it, is varied in different cases. Each case is experimented with different sizes of system slot, from 1 to 10 percent of a partition slot. There is a predictable linear relationship between the number of slots allocated to a partition and the processor availability. The partitions can therefore be configured to deliver a predictable level of service. Moreover, as can be seen in Figure 5.2, the maximum availability of the processor in case of temporal partitioning is less than 100%. This is due to the CoMik slot overhead.

Depending on an application requirement, it may or may not possible to run the application with particular size of the partition. Therefore, care must be taken at design time when dimensioning the TDM table. For instance, shorter TDMslots allow for higher throughput of partition context switches, enabling lower application response times, but increase context switch overhead. The proposed lower bound on the availability function may be used to analyze the timing properties of applications and to perform the schedulability analysis for different task scheduling policies at design time. As an example, here, we propose a schedulability analysis for the cases when the preemptive fixed-priority task scheduler is used.

The traditional response time analysis method of real-time applications using the fixed-priority scheduling is well known from the literature [47]. Following this method for our platform, we propose the worst-case response time calculation method as follows.

$$wcrsp_{ji} = C_i + \left(\sum_{k=1}^{i-1} \left\lceil \frac{wcrsp_{ji}}{T_k} \right\rceil \times C_k \right) + \left\lceil \frac{wcrsp_{ji}}{T_S} \right\rceil \times lbt_j \times \frac{S_p}{S} + \left\lceil \frac{wcrsp_{ji}}{S} \right\rceil \times S_m \quad (5.7)$$

For solving this equation, we refer to the existing approaches for traditional methods in the literature [47]. An application is schedulable with a fixed priority scheduler if and only if for all its tasks the condition of $A_j^l(wcrsp_{ji}) < A_j^l(d_{ji})$ holds true.

Using the above presented formulas, developers could analyze the worst case timing behavior of applications when using different intra-application scheduling policies, before actually running the applications on the platform, and design the applications for the worst-case scenarios. On the other hand, having the timing requirements of an application, the system and each partition properties could be tuned in such a way that the application is schedulable in its partition on the composable system.

Using the proposed availability function and the response time analysis method, in what follows we setup a Matlab simulation environment to study the timing behavior of a number of applications.

5.1.1 Matlab Simulation

Based on the availability function introduced in Equation (5.1), an instance of the platform is modeled in Matlab to simulate the timing behavior of a number of applications. For this purpose, 20 applications each consists of five tasks are generated randomly with a modified version (to generate random periodic tasks) of TORSCHE toolbox [94]. Each application is schedulable on a dedicated system in which the application owns the processor exclusively. This corresponds to a dedicated, non-composable system. Here, we consider a system TDM with period of 12 slots where we composably run multiple applications. Starting with the minimum partition size of one slot, in each run, we increase the size of the partition with one slot resulting in performing 12 different runs.

To investigate the variations in timing behavior, we define the *responsiveness* of a task as $\left(\frac{rsp}{T}\right)$. This gives us a normalized metric to compare the timing behavior of the tasks in our randomly generated applications with different timing properties, i.e, deadline and period. If the responsiveness of a task is greater than one, the task will miss its deadline.

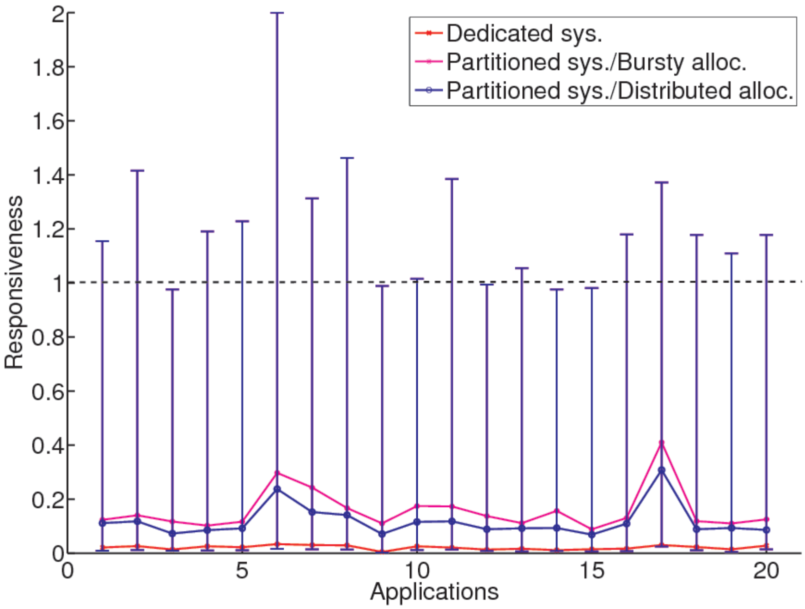
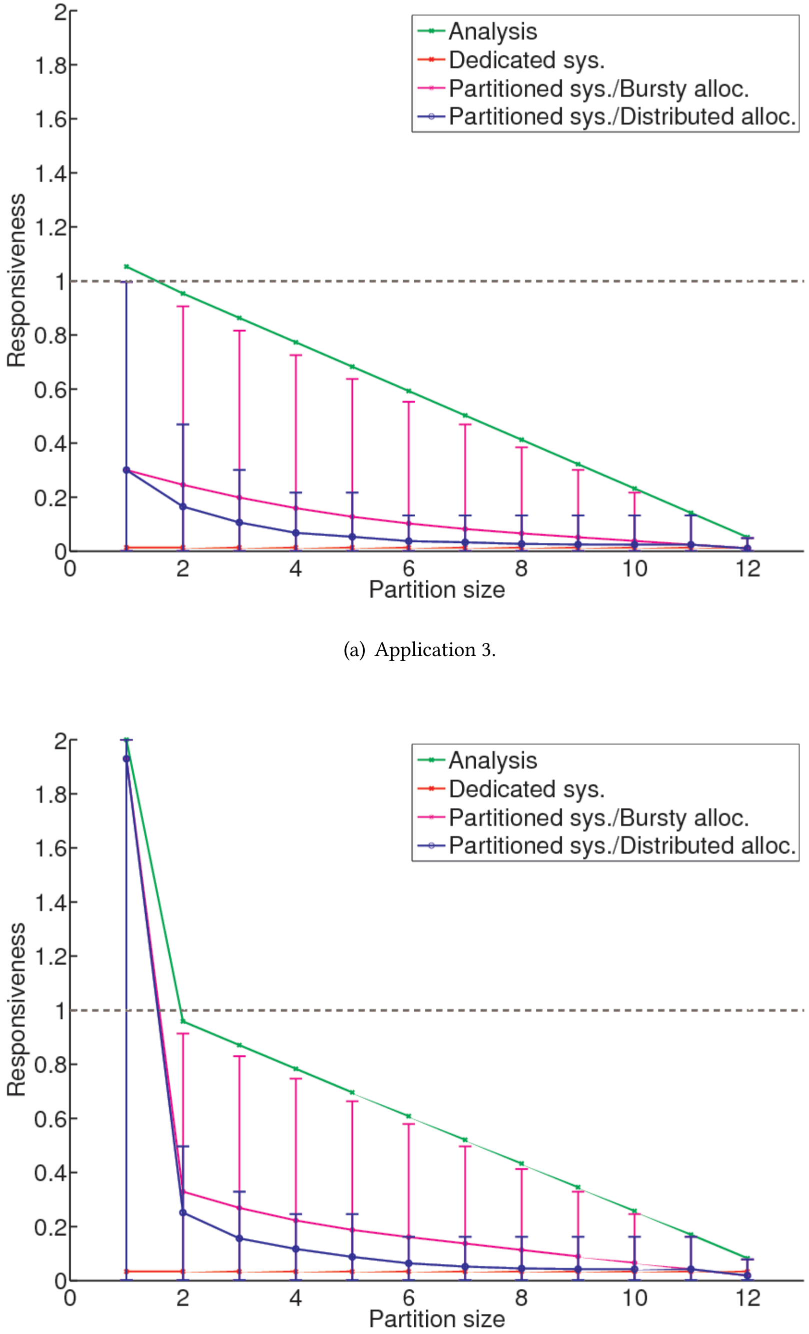


Figure 5.3: Average responsiveness of randomly generated applications, together with the min-max illustration of responsiveness in the partitioned-system distributed-allocation scenario.

Figure 5.3 illustrates the responsiveness of the applications in three different cases. The first case demonstrates the average of the worst-case responsiveness of the tasks in each application when executing on a dedicated system. For the other two cases we present the min-max bar and the average of the tasks' responsiveness in each application when executing on a partitioned system in the 12 different runs. In the second case the slot allocations to the partitions are bursty, while in the third case the allocations are as evenly distributed as possible. As expected, the results indicate that the average responsiveness of the tasks for all applications in both partitioned scenarios are always higher than the dedicated system scenario. This is a drawback of the partitioned system. The responsiveness in bursty allocations is also higher than the distributed allocations, since the applications experience longer bursty blocking time. Finally, we can observe that in this set of applications, 80% of the applications could not meet their timing requirements on the partitioned system regardless of their partition size as the maximum responsiveness of (at least one of) their tasks (is) are greater than one.

Figure 5.4 illustrates the responsiveness of the tasks of two applications, namely Application 3 & 6, when executing with different partition sizes. According to the Matlab simulation results, Application 3 is schedulable on the partitioned system as the maximum responsiveness of the its tasks are smaller than one in all runs, as illustrated in Figure 5.4(a). While, Application 6 is not schedulable in the run with the partition size of one, as illustrated in Figure 5.4(b). However, using Equation (5.7), the results of the worst-case responsiveness analysis of the both applications indicate that none of them are schedulable with the smallest partition size. In Figure 5.4, the differences between



(a) Application 3.

(b) Application 6.

Figure 5.4: *Average responsiveness of two randomly generated applications.*

the maximum responsiveness results of bursty allocations obtained from the Matlab simulation and the analysis results are because of not long enough simulation time during which the worst-case blocking situation between the tasks of the application could not occur. Moreover, as expected, by increasing the size of the applications' partitions the average responsiveness of the tasks is decreased.

5.2 Composability & Mixed Models of Computation

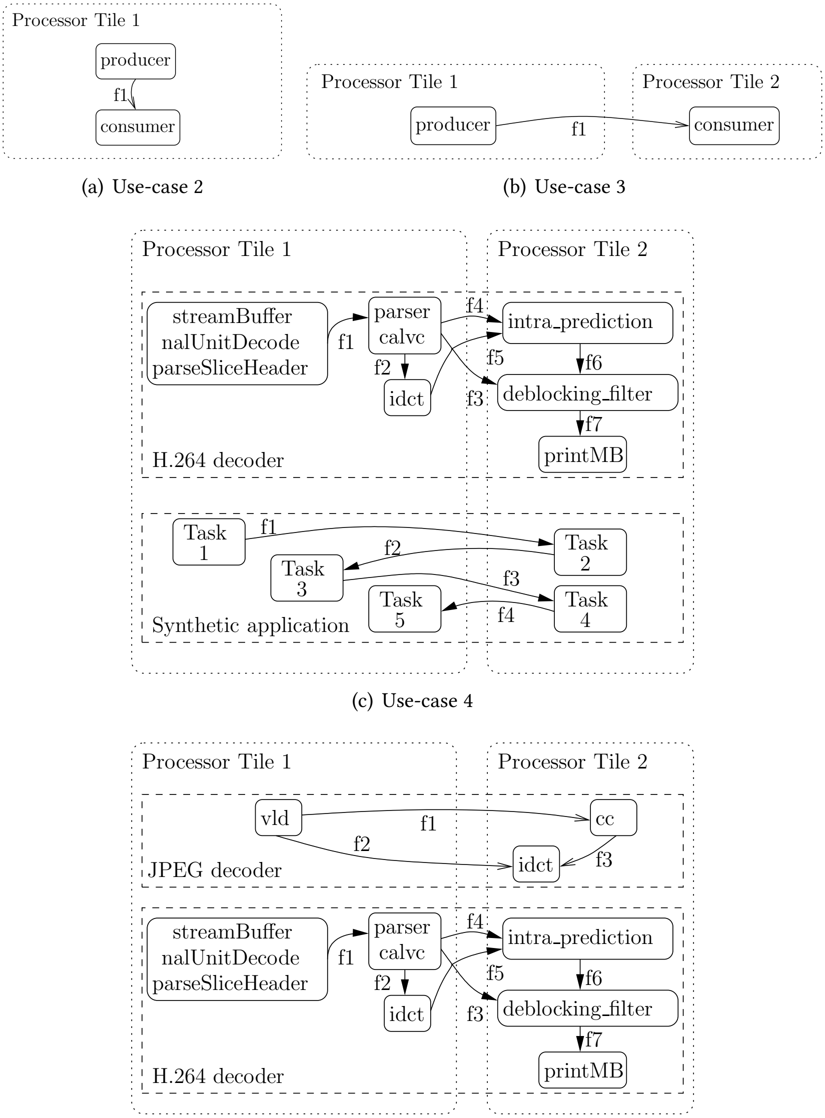
CoMik is designed and implemented as a composable micro-kernel that guarantees isolated and independent temporal (and functional) behavior of concurrent applications. Each application runs in a dedicated partition directly or indirectly on top of an OS library. In order to support execution of multiple applications implemented with different models of computation, CompOSe offers our unified model of execution. In this section, we demonstrate how the applications with different models of computation are implemented and executed using CompOSe on top of CoMik micro-kernel in the CompSOC platform. We also visualize the composability property of this system. Moreover, the performance of using preemptive and cooperative scheduling classes for some example use-cases of data-drive applications are discussed based on the empirical results.

All the experiments are performed on an FPGA prototype of the platform. A VHDL implementation of the CompSOC platform running with clock frequency of 120 MHz is prototyped on a Xilinx ML605 FPGA board [100]. This prototyped instance of the platform contains two processor tiles interconnecting with the an on-chip network. Each processor tile includes two RDMAs to be used for out-tile communications. Moreover, an external SDRAM is also available. CoMik is configured to use a system slot duration of 4096 cycles and a partition slot duration of 100000 cycles.

We set up five use-cases.*Use-case 1* has two applications.The first application is a workload of an electronic stability control (ESC) application having four periodic tasks, and the second one is a tick generating application having three periodic tasks. This use-case execute on one processor tile of the platform. We use this use-case to demonstrate concurrent, composable execution of multiple applications (in this case two time-driven ones) on the platform. Note that each application runs in its own partition, uses its own copy of CompOSe library, and therefore two preemptive task schedulers and one TDM partition scheduler are present.

The *use-cases 1 & 2* use a simple synthetic data-driven application in a one-tile and two-tile mappings on the platform, respectively, as shown in Figure 5.5(a) and 5.5(b) . This data-driven application is composed of two tasks, one producer and one consumer, communicating via a FIFO. These use-cases are to demonstrate the basic insights into how data-driven models of computation behave on the CompSOC platform when implemented in our model of execution. Moreover, by using preemptive and cooperative task schedulers we study the performance trade-off involved in using each class of the schedulers.

To investigate more in depth the trade-offs in implementing different models of computation with the model of execution, we set up the *use-case 3 & 4*, as illustrated in Figure 5.5(c) and 5.5(d). The fourth use-case consists of two applications, namely, a synthetic



(a) Use-case 2

(b) Use-case 3

(c) Use-case 4

(d) Use-case 5

Figure 5.5: Data-driven application use-cases.

one and and H.264 decoder. The synthetic application is a five-task application implemented with both KPN and CSDF models of computation, where each task has the same computation workload. The H.264 decoder is a six-task application, also realized in the two models of computation.

The *use-case 5* consists of the H.264 application and a JPEG decoder application. We use this case to verify the composability of the system when multiple applications execute concurrently on the platform.

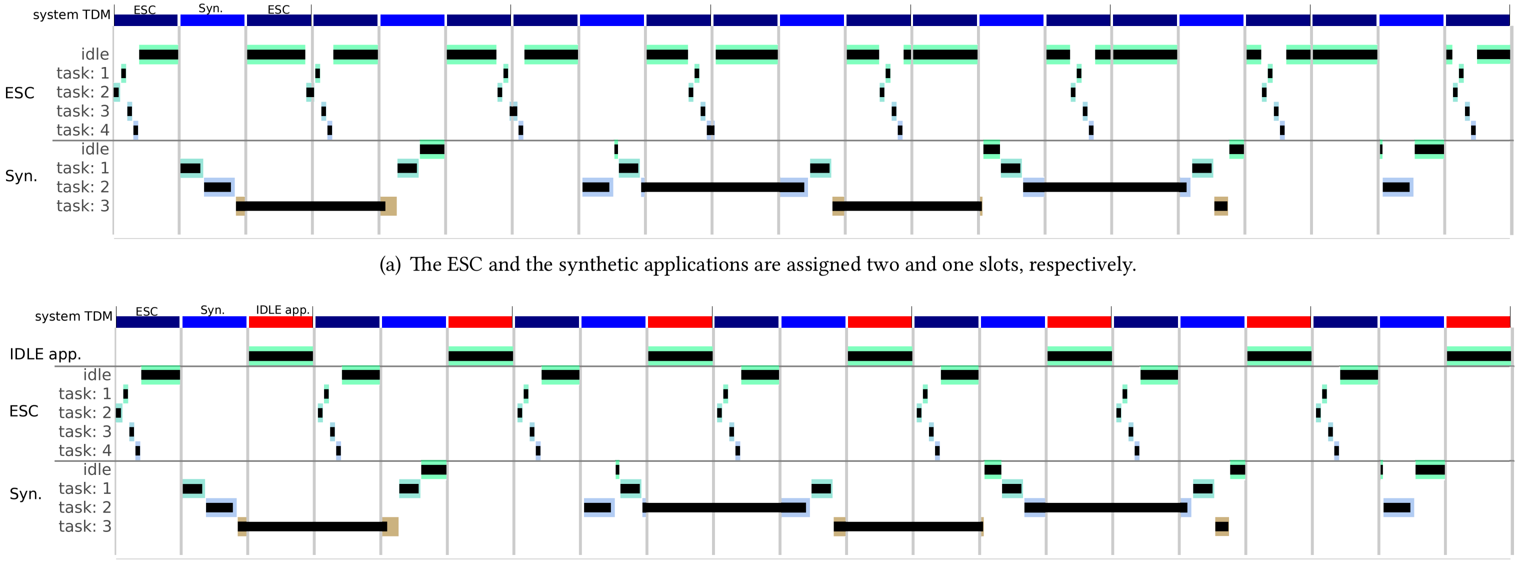
In the rest of this section, we demonstrated and discuss the experiment results of each use-case.

Composability: Use-case 1

The time-driven applications in use-case 1 use fixed-priority task scheduling in their own partition. Figure 5.6 illustrates the schedule traces of two different runs measured on the FPGA prototype with the inter-application TDM and intra-application fixed-priority scheduling of the applications. The TDM period of the inter-application scheduler is 3 slots. In the first run, the ESC and the synthetic applications are assigned two and one slots, respectively, as illustrated in Figure 5.6(a). In the second run, each application is assigned one slot, and the last slot is allocated to the *idle* application, as illustrated in Figure 5.6(b). In the figures, the vertical gray bars represent the OS slots, and between each two consecutive bars is an application slot. The colored boxes illustrate the execution times of the tasks, and the black boxes illustrate the duration between the logical execution time (as defined in Section 2.4.2) of the tasks.

The tasks of the ESC application are scheduled in its slots fully independent of the synthetic application's tasks. In the run illustrated in Figure 5.6(a), the longest blocking time for the ESC application is one slot and for the synthetic application is two-slot, i.e., the ones that are allocated to the ESC application. While, in Figure 5.6(b), the longest blocking time for the both application is two-slot, namely, the one slot that is allocated to the application itself, plus the slot of the idle application. The workload of ESC tasks is much less than an application slot size and all the tasks could always finish in the same slots that they start in. This is not the case for the tasks of the synthetic application. For example, the first execution of the task 2 starts in the second slot and it finishes in the fifth slot, and consequently, the response time of the task is more than the case that it runs on a dedicated system.

To illustrate the composability of the system, we compare the response time of the tasks in the two runs, as illustrated in Figure 5.7. The response times of ESC tasks are increased in the second run because the ESC application is allocated one less slot than the first run. As it is also expected from Equation (5.3), the longest blocking time of the ESC application in the second run is greater than in the first run, and therefore, its tasks start executing later (farther to its release time) and subsequently finish later. In spite of the variations in the timing behavior of the ESC tasks in these two runs, the response times of the synthetic tasks are remained unchanged. This observation shows that the timing properties of an application are not affected when the behavior of other applications is modified, and therefore, the applications are independent and the system is composable.



(a) The ESC and the synthetic applications are assigned two and one slots, respectively.

(b) The ESC and the synthetic applications are each assigned one slot.

Figure 5.6: *Schedule trace of the applications running on an FPGA prototype.*

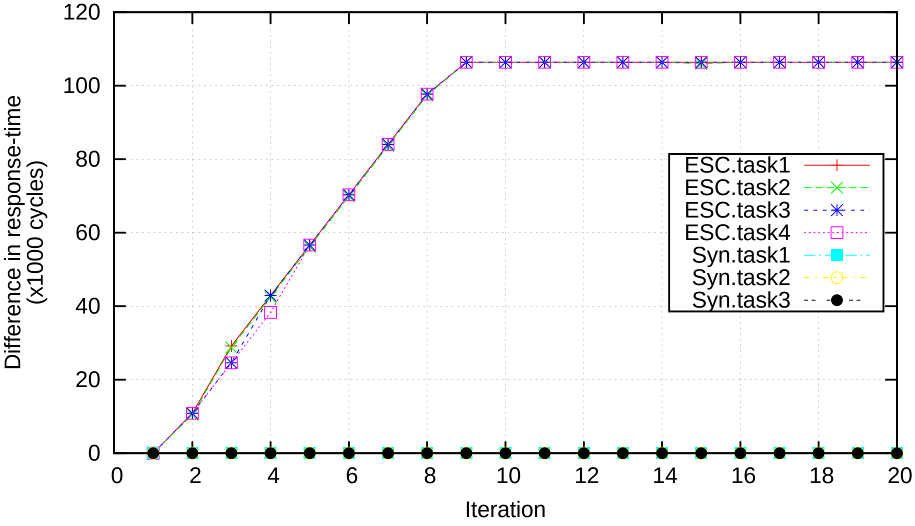


Figure 5.7: *Difference between response-time of the tasks in two runs, where the processor allocation to ESC application is changed.*

Mixed Model of Computation Support: Use-case 2 & 3

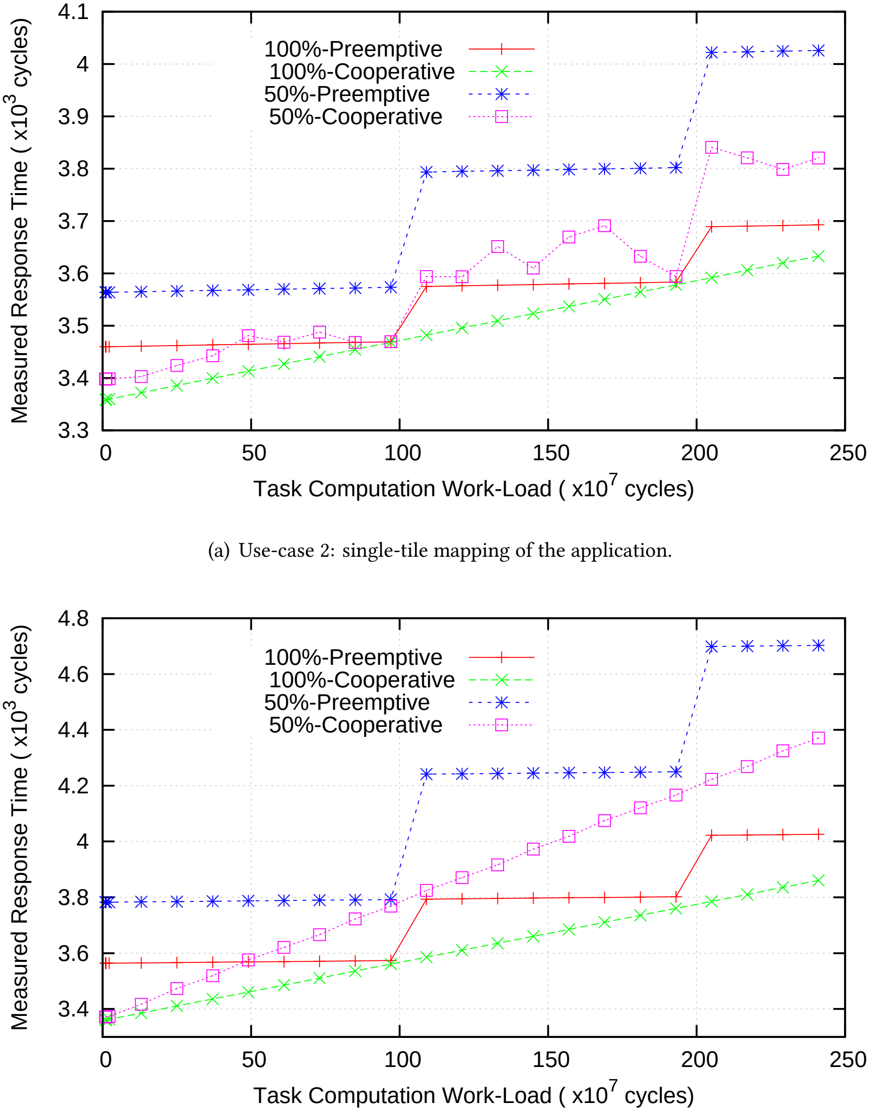
In use-case 2, the entire application is mapped on one processing tile, and in the third use-case, the producer task is mapped on the first processor tile and the consumer task is mapped on the second one.

Figure 5.8(a) and Figure 5.8(b) present the performance of the application in the two use-cases, when the application's share of the processor is changed from 100% to 50%. For each set-up, we perform one experiment with preemptive task scheduling and one with cooperative task scheduling.

In both use-cases, the application performs better if the tasks are scheduled cooperatively than preemptively. This is due to the fact that the application waits for the scheduler interrupt if the application finishes earlier. Whereas, in cooperative scheduling case, the task scheduler immediately schedules the next task. The difference between the results of preemptive and cooperative scheduling in Figure 5.8(a) and Figure 5.8(b) for each use-case, shows the wasted processor time in case of preemptive scheduling.

Regardless of the fact that the communication overhead increases in use-case 3, comparing to use-case 2, better performance results are obtained, as illustrated in Figure 5.8(b). This is because of executing the tasks in parallel on the two processors on each of which each task owns the whole partition time exclusively.

The finishing time graph for the cooperative scheduling case, when the application has 50% of the processor time, is presented in Figure 5.8(b). Unlike when the application has 100% of the processor time, this graph is not perfectly linear. This is due to the waiting time caused by the fact that the application on the second tile is swapped out of the processor. In this time, the consumer task cannot start processing the data and it should wait until its next partition slot. The waiting time directly depends on how the TDM



(a) Use-case 2: single-tile mapping of the application.

(b) Use-case 3: two-tile mapping of the application.

Figure 5.8: *Two use-cases of a simple synthetic application mapped on (i) one processor tile, and (ii) two processor tiles; illustrating finishing time for 10 iterations of each use-case.*

slots of an application's partitions on the resources, in this case the processor and the on-chip interconnect, are aligned. Ignoring the allocation on the on-chip interconnect as it is not changed in the two experiments, 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 it is linear in the case of Figure 5.8(a).

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 affect also explains the better performance in some cases in Figure 5.8(b) even if the task work-load increases. When designing a real-time application, the worst-case TDM slots miss-alignment should be taken into account to estimate the bounds on timing properties of the applications.

Preemptive vs. Cooperative Scheduling: Use-case 4 & 5

Figure 5.9(a) presents the performance of the synthetic application for various partition slot sizes, in use-case 4. Except for small slot sizes, cooperative task scheduling leads to better performance, in both KPN and CSDF, because the cooperative scheduling is work-conservative and utilizes the entire partition slot. There is a large waste of partition time in the case of preemptive scheduling. Here, for CSDF the performance differences between preemptive and cooperative scheduling are minor.

In the case of KPN, preemptive scheduling performs poorly, because the status of a task is not known when the task is selected and when the task is blocked the entire time until the next interrupt, when the scheduler is being invoked, is wasted.

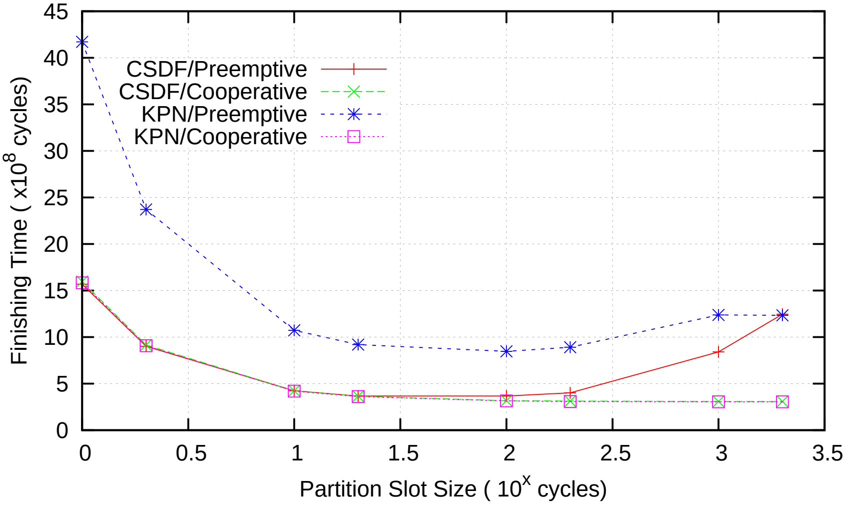
Figure 5.9(b) illustrates the performance of the H.264 modeled in KPN and CSDF. Similar to the synthetic application, in H.264 cooperative scheduling leads to better performance, small slot sizes have large overhead, and the KPN with preemptive scheduling has the worst performance.

In use-case 5, to verify the composability of the system when multiple applications execute on the platform, we present two scenarios for this use-case when the task scheduling class is changed for one of the applications and for the other one is kept unchanged.

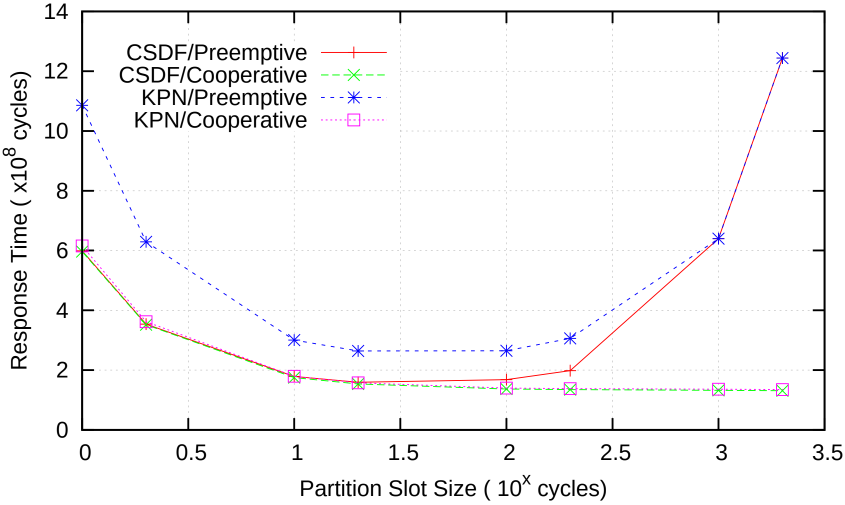
First, we execute the H.264 with the cooperative task scheduling, while the JPEG executes once with preemptive task scheduling in and another with cooperative task scheduling. Figure 5.10(a) presents the response time difference between the results of the experiments. The JPEG application shows no response 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 preemptive task scheduling and another with cooperative task scheduling, for the case when the JPEG task scheduling is kept unchanged. As expected, Figure 5.10(b) 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.

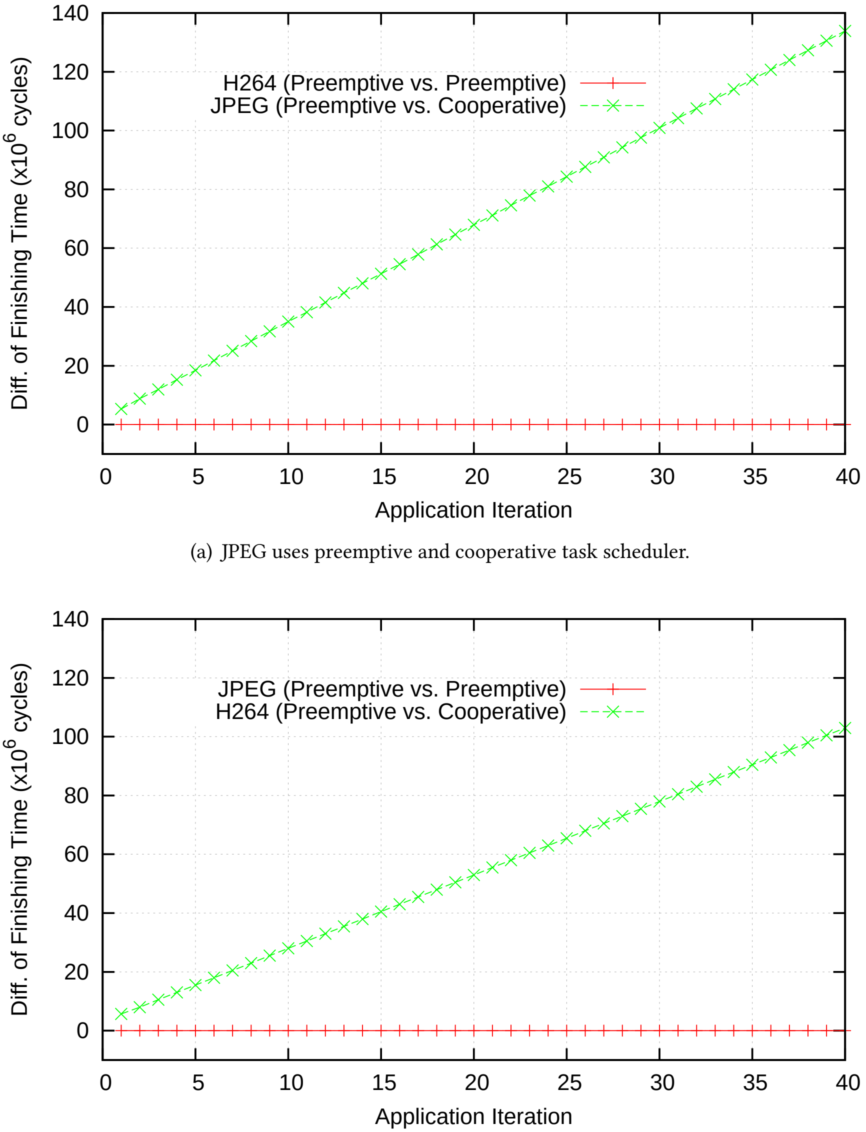


(a) The synthetic application



(b) H.264 video decoder.

Figure 5.9: *Synthetic and H.264 application on a two-Tile MPSoC Platform; illustrating finishing time for 20 iterations of each application, when they are realized in KPN and CSDF models of computation, and cooperative and preemptive task scheduler used to execute each model.*



(a) JPEG uses preemptive and cooperative task scheduler.

(b) H.264 uses preemptive and cooperative task scheduler.

Figure 5.10: *The finishing time difference between two execution scenarios of JPEG and H.264 applications. In each scenario task scheduling of one of the applications is changed from preemptive to cooperative.*

5.3 Summary

In this chapter we have demonstrated empirically how the platform executes mixed criticality applications realized with either of data- or time-driven models of computation. For this, we studied the specific features of the platform as (i) predictability, (ii) composability, and (iii) support for mixed models of computation.

The study has been performed using two experiment setups. First, Matlab simulation of the virtual platforms is used to show predictable temporal behavior of CoMik in partitioning the processor tiles for time-driven models of computation. For this experiment, we first formalized the timing properties of the composable platform, and proposed a response time analysis method for time-driven applications using fixed-priority task scheduling. Then, we simulated the timing behavior of a number of randomly generated applications on a simulating instance of the platform. With this, we have shown how the behavior of the applications changes predictably when the size of their allocated partitions are changed.

Second, FPGA emulation of a prototyped instance of the platform when executing different application use-cases is used. With this, we have shown the composability and support for mixed models of computation features of the system.

CHAPTER 6

Conclusions

THIS chapter concludes this dissertation with respect to the main contributions and the implementation of the solution described throughout this dissertation. It also proposes several opportunities for future research work.

6.1 Contributions

This dissertation answers the major research question of how to design and execute multiple (real-time) applications concurrently on an embedded system, given that the applications are realized with different models of computation and they have different level of time-criticality. This question has been raised as a result of two design challenges that exist in the field of embedded systems. These challenges have been addressed as: (i) realizing strongly composable Virtual Platforms (VPs) for the mixed-criticality embedded systems, and (ii) proposing an abstract execution layer in implementation of applications expressed with different models of computation on the VP in a unified manner.

The solution to this question is to use virtualization technology to make the mixed-criticality systems predictable and composable. This approach creates a VP for every application by virtualizing all the resources that are involved in the execution of the applications. For this, the partitioning technique is used. This technique is applied to different resources of the platform to partition them temporally or spatially. In the context of mixed-time criticality system, our focus is on temporal partitioning of the resources. For this, a complete platform-based approach is used to mitigate the complexity of a mixed-criticality system stack by proposing a multi-layered virtualization software platform on top of a Multi-Processor System-on-Chip (MPSoC) hardware architecture.

In the hardware platform, each resource is specially designed and implemented for the purpose of guaranteeing predictable and composable execution of applications with mixed-time criticality. CompSOC is a System-on-Chip (SoC) template developed following composability and predictability paradigm. This template is used as the basis of the hardware architecture.

Composable temporal partitions of the processor tile are created by using Time Division Multiplexing (TDM) arbitration scheme that provides a guarantee on when the service is exactly available to a partition. This scheme prevents the interference of other partitions, while every application executes in a dedicated partition. Applying the composable partitioning technique to all the resources involved in executing an application in a processor tile, a partition is formed. A VP is then a set of these partitions allocated to one application from all over the platform.

On top of the hardware platform, a minimum privileged software layer is designed to provide the essential services of partitioning for the purpose of virtualization. This layer is realized in the form of a microkernel, namely CoMik. CoMik creates, controls and schedules processor tile partitions. It executes an application in its partition by virtualizing the software hooks and offering an Application Programming Interface (API) so that the application could use its allocated resources.

Furthermore, a model of execution is proposed to fill the gap of execution abstraction between the models of computation's semantic and the platform's primitive operations. The model of execution categorizes the execution operations as: (i) computation, (ii) communication, and (iii) scheduling operations.

The model of execution is implemented in form of CompOSe Operating System (OS) library which is instantiated in every partition of the VP created by CoMik for an application. CompOSe is designed in a number of software units for: (i) providing the *main*, *interrupt handler*, and *exception* handler software hooks required by CoMik from a partition, (ii) implementing the execution primitives of the model of execution using the API provided by CoMik, and (iii) giving model of computation specific support to the application by providing Software Containers (SCs). CompOSe is implemented in such a way that it does not introduce any uncertainty in executing an application on top of the CoMik microkernel and therefore it complies with predictability and composability of the system.

Finally, the predictability, the composability, and the support of multiple models of computation are demonstrated empirically by using two major experiment setups: (i) a Matlab simulation of virtual platform, that investigates predictable temporal behavior of CoMik in partitioning the processor tiles; (ii) an FPGA prototype of the platform, that studies the composability property and support of multiple models of computation. For this, a number of use-cases consisting of real and synthetic applications execute on a MPSoC instance of the CompSOC platform.

6.2 Future Research Opportunities

The work that has been presented in this dissertation can be extended for further improvement. Here, we elaborate on two of the possibilities with this regard.

- In the current system, processor temporal partition are created by uniforming arbitrating the utilization time of the processor using the TDM scheme. When a number of consecutive time slots are allocated to one partition (an eventually to one application), since each slot comes with a fixed overhead of the CoMik slot at the start, it results in a non necessary overhead and consequently under-utilizing the system. Instead on such uniform TDM arbitration, a nun-uniform scheme can be used to improve this situation. This scheme defines a finest possible grain slot sizes, namely slot unit, for the system. When allocating slots to the partitions, it initially analyzes the allocations using the slot units. In the second step, where a number of consecutive unit slots are allocated to one partition, these slot can be merged together and form a bigger size partition slot with just one CoMik slot overhead at the start. This scheme is fully compliant with the composability property of the system, as no matter what the size of an slot is, the partitions are fully isolated and independent from each other. The complexity of the solution relies in the fact that the CoMik has to then schedule possibly different size slots in each partition scheduling time which is also feasible using the current design of Partition Control Block (PCB) and Timer-centric Interrupt and Frequency Unit (TIFU).

- Sizing partitions for every application is a challenging task for designers. In the case that the application is expressed in a data-driven model of computation, there exists automatic tool flows to perform analysis of the application model and come up with the proper sizing of partitions such as the wok in [6]. There is a research opportunity in the context of extending this work in order to integrate the proposed system analysis formalization in this dissertation with application analysis methods so that the process of creating a proper size partition for a time-driven application can be done automatically. In this way, the design flow can come up with possibly various partition sizes and the system slot allocation. This brings more flexibility when integrating multiple applications with mixed-criticality on the platform.

APPENDIX A

Software-Based Interrupt Virtualization

CONSIDERING the architecture for the processor tile of the CompSOC platform without having TIFU as depicted in Figure 2.2. In such a architecture, the only partition-level interrupt that may be supported is the timer interrupt. But, only CoMik uses Clock Control Module (CCM) for the system-level interrupt, and if an application executing in a partition needs to use the timer interrupt functionality, the application would interfere with CoMik in using CCM. Consequently, the system interrupt may be affected, and the system composability is compromised. In order to prevent this, in this section we propose a software-based approach to virtualize CCM and enable the system- and partition-level timer interrupts to co-exist.

We propose a software-based technique to virtualize the CCM physical timer to be used for the purpose of generating both system and partition timer interrupts. The system interrupt timer has to be active in the whole system lifetime to implement composable temporal partitions. Thus, when a partition needs to use the timer interrupt, there are logically two virtual timers active in the partition time. The virtualization technique realizes these two logical timer interrupts while preserving the system composability. This means using this technique the timing properties of the other partition slots, i.e., start and end time of the slots, are not altered.

This technique is implemented in a kernel-mode function denoted as Programmable Interrupt Timer (PIT) function. This function is called inside the partition interrupt handling routine to program CCM for the next timer interrupt. Figure A.1 illustrates the operational time-line of a partition interrupt handling routine when a timer interrupt is occurred at time t. After a small delay (which is the result of executing a critical section or a multi-cycle instruction), the processor starts executing the operations to handle the partition interrupt. When the interrupt is handled, before returning from the interrupt handler, the Programming Timer Interrupt (PTI) function executes to program the timer for the next interrupt.

The value for the next partition interrupt is typically given relatively to the end of the previous interrupt handling interval, and therefore, the overhead of handling the

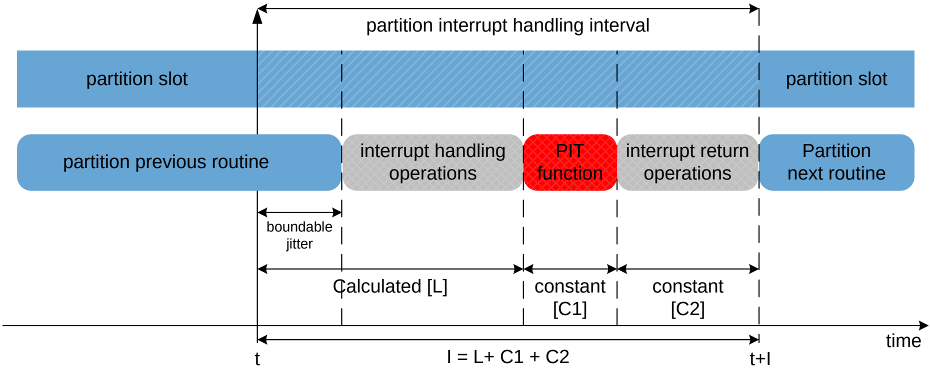


Figure A.1: The General operational time-line of an application in a partition interrupt handling interval.

interrupt has to be taken into account by the PTI function to calculate how much time of the partition current slot is left. Such an overhead is not constant. Thus, the PTI function calculates the time that it has been taken to execute the handling operations. This is L time units, as depicted in Figure A.1. Moreover, the PTI function itself and the operation to return from the handler are implemented with constant execution times of $C1$ and $C2$ time units, respectively. Therefore, the partition interrupt handling interval would be $I = L + C1 + C2$ time units.

In the rest of this section, we present the detailed implementation of the PTI function in virtualizing the system and partition timer interrupts.

Implementation

The main idea behind the software-based technique is that the PTI function updates all the two logically active timer interrupts and program the physical timer, i.e., CCM, with the value of the earliest one. The earliest interrupt may be either a system interrupt, or a partition interrupt of the running partition (application). Between these two logical interrupts, precedence is given to system interrupts, such that they always occur at fixed moments in time and the system composability is not invalidated. The main challenges here are to accurately calculate the moment when the timer interrupts should occur, and to keep track of partition interrupt handling intervals so that the moment when each partition slot starts remains unchanged.

The partition interrupts (for example in a time-triggered application) are either periodic or aperiodic in the partition time, and the interrupts are not perfectly aligned with the system interrupt. In order to keep the timing information and to update it as the time progress, the data structures has to be supported in the CoMik Control Block (CCB) and the PCB. The size of the CoMik (microkernel) slot and the partition slots, denoted as `mk_slot` and `par_slot`, respectively, are part of the system properties and kept in the CCB.

Table A.1 presents the extensions of additional data structures to the control blocks, that are required by the PTI function to calculate the time of next interrupts as follows:

Table A.1: *Additional data structure required for Software-based interrupt virtualization.*

SW Control Blocks	Data Structure
CCB	int mk_slot int par_slot int par_slot_rem
PCB	int pi_time int pi_time_rem int pih_interval int pih_interval_rem

- at every interrupt handling interval, the relative time to the end of the interval for the next partition interrupt time is given and denoted as `pi_time`.

- if the partition has already been preempted by the system interrupt before the next partition interrupt is occurred, the remaining time to the next partition interrupt time is calculated and denoted as `pi_time_rem`.

- assuming that the next partition interrupt time is occurred, the remaining time of the partition slot is calculated and denoted as `par_slot_rem`.

- the duration of the current partition interrupt handling interval is calculated and denoted as `pih_interval`. If a system interrupt is supposed to occur in the middle of the partition interrupt handling interval, it does not preempt the handling operations, however, the duration between the moment of the system interrupt until the end of the interval is denoted as `pih_interval_rem` to be considered at the beginning of the next slot that belongs to this partition.

The time for the partition timer interrupts is typically given in the system (real world) time, and it is need to convert this time into a moment in the partition virtual time. To do this, PTI function calls a function, denoted as `conv_st_to_pt`, that implements the conversion of a time moment in the system time to a moment in the partition time. Such a conversion directly depends on the partition parameters and the availability of the processor to the partition in the given partition interrupt time.

Listing A.1 details the implementation of the PTI function using the data structures presented in Table A.1. The PTI function calculates the new value of `par_slot_rem` by deducting the sum of the next `pi_time` and the current `pih_interval` from the remaining partition slot duration. According to the value of `par_slot_rem`, three different scenarios for the next interrupt are possible, as illustrated in Figure A.2. Following one of these scenarios, as presented in Listing A.1, the PTI function sets the timer with the proper value for the earliest interrupt in the future.

In the first scenario, after *current_int* interrupt has occurred, the partition interrupt handling interval can finish its entire operations and set the next timer interrupt, i.e., *next_int*, for a partition interrupt at a time before the next system interrupt. In this case, the `par_slot_rem` is greater than the current `pih_interval` plus the `pi_time`.

```
/* INPUT:  CCB, current PCB */
/* OUTPUT: Sets proper timer interrupt value, and updates all the data structures
     presented in~\reftab{data_struct} */

   int pih_interval = calculate_pi_interval();
   int pi_time_rem_virtual = conv_st_to_pt(pi_time_rem);
   int par_slot_rem -= (pih_interval + pi_time_rem_virtual);
   int pi_time_temp;

IF(par_slot_rem > 0) {
/* Scenario 1 */
     pi_time_temp = pi_time_rem_virtual;
     pi_time_rem = pi_time;
     pih_interval_rem = 0;

} ELSEIF (par_slot_rem <= 0) {
     pi_time_temp =  pi_time_rem_virtual + par_slot_rem;

     IF (pi_time_temp < 0) {
/* Scenario 2 */
       pih_interval_rem = -task_interval_temp;
       pi_time_temp = 0;
     }{
/* Scenario 3*/
       pih_interval_rem = 0;
       pi_time_rem = pi_time_rem - pi_time_temp;
     }
     par_slot_rem = par_slot;
}

   /* Set the next timer interrupt */
   program_CCM(pi_time_temp);
```

Listing A.1: *Pseudo code representation of the PTI function.*

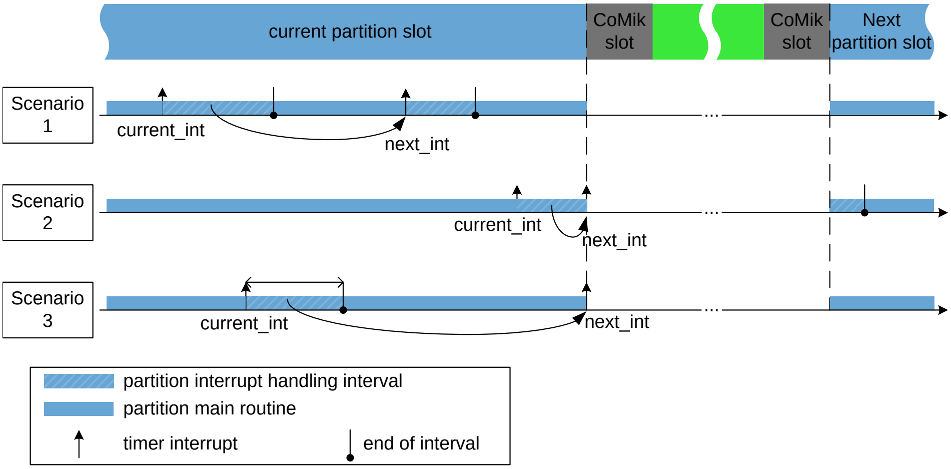


Figure A.2: *Three possible scenarios of programming the timer interrupt.*

In the second scenario, after *current_int* interrupt has occurred, the partition interrupt handling interval cannot finish its entire operations before the next interrupt *next_int* which is the periodic system interrupt. This is because the `par_slot_rem` is less than the current `pih_interval`. In this case, we let the partition interrupt handling operations to be finished, and therefore the interval extends over the CoMik slot. The CoMik slot has fixed size and it is designed in such a way that it can absorb the worst case duration of the partition interrupt handling intervals for all the running applications and its size remains unchanged. The extended time of the interval over the CoMik slot, or in other words, the remaining time of the interval from the current partition slot, is calculated in `pih_interval_rem`. This remaining time is implemented at the beginning of the next partition slot so that the application running in this partition does not experience a shorter interrupt handling interval. If *current_int* happens exactly at the moment of the periodic system interrupt, then the entire partition interrupt handling interval is considered in `pih_interval_rem` and implemented in the next slot that belongs to this partition.

In the third scenario, when the current interrupt *current_int* occurs, the partition interrupt handling interval can finish its entire operations, but the system interrupt is before the next partition interrupt, i.e., *next_int*. This is because the `par_slot_rem` is less than the current `pih_interval` plus the `pi_time`. The remaining partition interrupt time, i.e., `pi_time_rem`, is therefore implemented in the next slot that belongs to this partition.

In summary, the proposed software-based technique virtualizes CCM by implementing the PTI function that manages the co-existence of system and partition timer interrupts in each partition so that the timing properties of partition slots remain unchanged and the system composability is preserved. The software-based approach is generic in the sense that it works with a minimum hardware requirement, i.e., only one timer, while the hardware-based solution requires TIFU to extend the interrupt virtualization functionality. On the other hand, the software-based approach virtualizes only the timer interrupts, whereas, the second approach includes also the support of external interrupts for the partitions.

Bibliography

[1] *Aguiar, A., and Hessel, F.* Embedded systems' virtualization: The next challenge? In *International Symposium on Rapid System Prototyping* (2010), pp. 1–7.

[2] *Akesson, B., and Goossens, K.* *Memory Controllers for Real-Time Embedded Systems*, first ed. Embedded Systems Series. Springer, 2011.

[3] *Akesson, B., Goossens, K., and Ringhofer, M.* Predator: A predictable SDRAM memory controller. In *IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)* (2007), pp. 251–256.

[4] *Akesson, B., Molnos, A. M., Hansson, A., Ambrose Angelo, J., and Goossens, K.* Composability and predictability for independent application development, verification, and execution. In *Multiprocessor System-on-Chip*. November 2010, pp. 25–56.

[5] *Akesson, B., Steffens, L., and Goossens, K.* Efficient service allocation in hardware using credit-controlled static-priority arbitration. In *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)* (2009), pp. 59–68.

[6] *Akesson, B., Stuijk, E., Molnos, A., Koedam, M., Stefan, R., Nelson, A., Nejad, A. B., and Goossens, K.* Virtual platforms for mixed time-criticality applications: The CoMPSoC architecture and SDF3 design flow.

[7] *Almeida, L., and Pedreiras, P.* Scheduling within temporal partitions: response-time analysis and server design. In *International Conference on Embedded Software (EMSOFT)* (2004).

[8] *Auge, I., Petrot, F., Donnet, F., and Gomez, P.* Platform-based design from parallel C specifications. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 24*, 12 (2005), 1811 – 1826.

[9] AUTOSAR (AUTomotive Open System ARchitecture). `http://www.autosar.org/`.

[10] *Axer, P., Ernst, R., Falk, H., Girault, A., Grund, D., Guan, N., Jonsson, B., Marwedel, P., Reineke, J., Rochange, C., Sebastian, M., Hanxleden, R. V., Wilhelm, R., and Yi, W.* Building timing predictable embedded systems. *ACM Transaction on Embedded Computing Systems 13*, 4 (Mar. 2014), 82:1–82:37.

[11] *Bamakhrama, M., and Stefanov, T.* Hard-real-time scheduling of data-dependent tasks in embedded streaming applications. In *International Conference on Embedded Software (EMSOFT)* (2011), pp. 195–204.

[12] *Bekooij, M., Wiggers, M., and van Meerbergen, J. L.* Efficient buffer capacity and scheduler setting computation for soft real-time stream processing applications. In *International Workshop on Software and Compilers for Embedded Systems (SCOPES)* (2007).

[13] *Broy, M.* Challenges in automotive software engineering. In *International Conference on Software Engineering (ICSE)* (2006), pp. 33–42.

[14] *Bui, D., Lee, E., Liu, I., Patel, H., and Reineke, J.* Temporal isolation on multiprocessing architectures. In *Design Automation Conference (DAC)* (2011).

[15] *Carpenter, J., Funk, S., Holman, P., Srinivasan, A., Anderson, J., and Baruah, S.* A categorization of real-time multiprocessor scheduling problems and algorithms. In *Handbook on Scheduling Algorithms, Methods, and Models* (2004).

[16] *Castrillon, J., Velasquez, R., Stulova, A., Sheng, W., Ceng, J., Leupers, R., Ascheid, G., and Meyr, H.* Trace-based KPN composability analysis for mapping simultaneous applications to MPSoC platforms. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2010), pp. 753–758.

[17] *Dyer, M., Platzner, M., and Thiele, L.* Efficient execution of process networks on a reconfigurable hardware virtual machine. In *Field-Programmable Custom Computing Machines (FCCM)* (2004), pp. 342 – 344.

[18] *Eker, J., Eker, J., Janneck, J. W., and Janneck, J. W.* A structured description of dataflow actors and its application. In *Technical Report UCB/ERL M03/13, EECS Department, University of California, Berkeley,* (2003).

[19] *Feng, X. A., and Mok, A. K.* A model of hierarchical real-time virtual resources. In *Real-Time System Symposium (RTSS)* (2002).

[20] *Ghosal, A., Henzinger, T. A., Kirsch, C. M., and Sanvido, M. A. A.* Event-driven programming with logical execution times. In *International Workshop on Hybrid Systems: Computation and Control (HSCC)* (2004).

[21] *Goossens, K., Azevedo, A., Chandrasekar, K., Gomony, M. D., Goossens, S., Koedam, M., Li, Y., Mirzoyan, D., Molnos, A., Nejad, A. B., Nelson, A., and Sinha, S.* Virtual execution platforms for mixed-time-criticality systems: the CompSOC architecture and design flow. *SIGBED Review 10*, 3 (2013), 23–34.

[22] *Goossens, K., and Hansson, A.* The Aethereal network on chip after ten years: Goals, evolution, lessons, and future. In *Design Automation Conference (DAC)* (2010), pp. 306–311.

[23] *Goossens, S., Akesson, B., Koedam, M., Nejad, A. B., Nelson, A., and Goossens, K.* The CompSOC design flow for virtual execution platforms. In *10th FPGAworld Conference* (September 2013).

[24] *Goossens, S., Kuijsten, J., Akesson, B., and Goossens, K.* A reconfigurable real-time SDRAM controller for mixed time-criticality systems. In *IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)* (2013), pp. 1–10.

[25] *Haid, W., Schor, L., Huang, K., Bacivarov, I., and Thiele, L.* Efficient execution of kahn process networks on multi-processor systems using protothreads and windowed fifos. In *Embedded Systems for Real-Time Multimedia (ESTIMedia)* (2009), pp. 35 –44.

[26] *Hansson, A., Ekerhult, M., Molnos, A., Milutinovic, A., Nelson, A., Ambrose, J., and Goossens, K.* Design and implementation of an operating system for composable processor sharing. *Microprocessors and Microsystems* (2011).

[27] *Hansson, A., and Goossens, K.* *On-Chip Interconnect with aelite: Composable and Predictable Systems.* Embedded Systems Series. Springer, Nov. 2010.

[28] *Hansson, A., and Goossens, K.* A quantitative evaluation of a network on chip design flow for multi-core consumer multimedia applications. *Springer Journal of Design Automation for Embedded Systems (DAEM) 15*, 2 (2011), 159–190.

[29] *Hansson, A., Goossens, K., Bekooij, M., and Huisken, J.* CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM Transactions on Design Automation of Electronic Systems* (2009).

[30] *Hansson, A., Goossens, K., and Rădulescu, A.* A unified approach to constrained mapping and routing on network-on-chip architectures. In *IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)* (2005), pp. 75–80.

[31] *Hansson, A., Subburaman, M., and Goossens, K.* Aelite: A flit-synchronous network on chip with composable and predictable services. In *Proceedings of the Design, Automation & Test in Europe Conference and Exhibition* (April 2009).

[32] *Heiser, G.* The role of virtualization in embedded systems. In *Workshop on Isolation and integration in embedded systems (IIES)* (2008), pp. 11–16.

[33] *Heiser, G.* Virtualizing embedded systems - why bother? In *Design Automation Conference (DAC)* (June 2011), pp. 901–905.

[34] *Heiser, G., and Leslie, B.* The OKL4 microvisor: Convergence point of microkernels and hypervisors. In *ACM Asia-pacific Workshop on Workshop on Systems (APSys)* (2010), pp. 19–24.

[35] *Henzinger, T. A.* Two challenges in embedded systems design: predictability and robustness. *Philosophical Transactions of the Royal Society 366*, 1881 (2008), 3727–36.

[36] *Henzinger, T. A., Horowitz, B., and Kirsch, C. M.* Giotto: A time-triggered language for embedded programming. In *PROCEEDINGS OF THE IEEE* (2001), Springer-Verlag, pp. 166–184.

[37] *Hergenhan, A., and Heiser, G.* Operating systems technology for converged ECUs. In *Embedded Security in Cars Conference (ESCAR)* (November 2008).

[38] *Holenderski, M., Cools, W., Bril, R. J., and Lukkien, J. J.* Multiplexing real-time timed events. In *International Conference on Emerging Technologies and Factory Automation (ETFA)* (2009).

[39] *Hur, J. Y., Stefanov, T., Wong, S., and Vassiliadis, S.* Customizing reconfigurable on-chip crossbar scheduler. In *IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)* (2007), pp. 210–215.

[40] *Hur, J. Y., Stefanov, T., Wong, S., and Vassiliadis, S.* Systematic customization of on-chip crossbar interconnects. In *International conference on Reconfigurable computing: architectures, tools and applications (ARC)* (2007), pp. 61–72.

[41] *Ito, M., and Oikawa, S.* Mesovirtualization: lightweight virtualization technique for embedded systems. In *International conference on Software technologies for embedded and ubiquitous systems (SEUS)* (2007), pp. 496–505.

[42] *Jantsch, A.* *Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation.* Electronics & Electrical. Morgan Kaufmann, 2004.

[43] *Jantsch, A., and Sander, I.* Models of computation and languages for embedded system design. *IEE Proceedings of Computers and Digital Techniques, - 152*, 2 (Mar 2005), 114–129.

[44] *Jerraya, A., and Wolf, W.* *Multiprocessor Systems-on-Chips (Systems on Silicon)*, 1 ed. Morgan Kaufmann, Oct. 2004.

[45] *Jerraya, A. A., Bouchhima, A., and Pétrot, F.* Programming models and HW-SW interfaces abstraction for multi-processor SoC. In *Design Automation Conference (DAC)* (2006).

[46] *John, R.* Partitioning in avionics architectures: Requirements, mechanisms, and assurance. Tech. rep., 1999.

[47] *Joseph, M., and Pandya, P. K.* Finding response times in a real-time system. *The Computer Journal 29*, 5 (1986).

[48] *Kahn, G.* The semantics of a simple language for parallel programming. In *The International Federation for Information Processing (IFIP) Congress.* 1974.

[49] *Kerstan, T., Baldin, D., and Groesbrink, S.* Full virtualization of real-time systems by temporal partitioning. In *Operating Systems Platforms for Embedded Real-Time applications (OSPERT)* (2010).

[50] *Kienhuis, B., Rijpkema, E., and Deprettere, E.* Compaan: deriving process networks from matlab for embedded signal processing architectures. In *IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)* (2000), pp. 13–17.

[51] *Kinebuchi, Y., Koshimae, H., Oikawa, S., and Nakajima, T.* Virtualization techniques for embedded systems. In *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA) – Work-in-Progress* (2006).

[52] *Kinebuchi, Y., Sugaya, M., Oikawa, S., and Nakajima, T.* Task grain scheduling for hypervisor-based embedded system. In *International Conference on High Performance and Communications (HPCC)* (2008).

[53] *Kopetz, H.* *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Real-Time Systems Series. Springer, 2011.

[54] *Kopetz, H., et al.* Compositional design of RT systems: a conceptual basis for specification of linking interfaces. *Proc. of ISORC* (2003).

[55] *Kumar, A., Mesman, B., Theelen, B., Corporaal, H., and Ha, Y.* Analyzing composability of applications on MPSoC platforms. *Journal of Systems Architecture* (March 2008).

[56] *Lee, E. A., and Parks, T. M.* Dataflow process networks. *Readings in hardware/-software co-design* (2002).

[57] *Lee, E. A., and Sangiovanni-vincentelli, A.* A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 17* (1998), 1217–1229.

[58] *Lickly, B., Liu, I., Kim, S., Patel, H. D., Edwards, S. A., and Lee, E. A.* Predictable programming on a precision timed architecture. Tech. rep., EECS Department, University of California, Berkeley, Apr 2008.

[59] *Lipari, G., and Bini, E.* Resource partitioning among real-time applications. In *Euromicro Conference on Real-Time Systems (ECRTS)* (2003).

[60] *Liu, I., Reineke, J., Broman, D., Zimmer, M., and Lee, E. A.* A PRET microarchitecture implementation with repeatable timing and competitive performance. In *International Conference on Computer Design (ICCD)* (October 2012).

[61] *Liu, J., and Lee, E. A.* Timed multitasking for real-time embedded software. *IEEE Control Systems Magazine 23* (2003), 65–75.

[62] *Martin, G.* Overview of the MPSoC design challenge. In *Design Automation Conference (DAC)*. 2006.

[63] *Masrur, A., Drossler, S., Pfeuffer, T., and Chakraborty, S.* VM-based real-time services for automotive control applications. In *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)* (2010).

[64] *Meenderinck, C., Molnos, A., and Goossens, K.* Composable virtual memory for an embedded soc. In *Proceedings of the 2012 15th Euromicro Conference on Digital System Design* (2012), Euromicro Conference on Digital System Design, Architectures, Methods and Tools (DSD), pp. 766–773.

[65] *Mok, A. K., Feng, X. A., and Chen, D.* Resource partition for real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2001).

[66] *Molnos, A., Nejad, A. B., Nguyen, B. T., Cotofana, S., and Goossens, K.* Decoupled inter- and intra-application scheduling for composable and robust embedded MPSoC platforms. In *5th Workshop on Mapping of Applications to MPSoCs & 15th International Workshop on Software and Compilers for Embedded Systems (Map2MPSoC/SCOPES 2012)* (May 2012), pp. 13–21.

[67] *Moreira, O., and Bekooij, M.* Self-timed scheduling analysis for real-time applications. *EURASIP Journal on Advances in Signal Processing 2007* (2007).

[68] *Moreira, O., Mol, J.-D., Bekooij, M., and van Meerbergen, J.* Multiprocessor resource allocation for hard-real-time streaming with a dynamic job-mix. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2005).

[69] *Moreira, O., Valente, F., and Bekooij, M.* Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor. In *International Conference on Embedded Software (EMSOFT)* (2007).

[70] *Nadesakumar, A., Crowder, R., and Harris, C.* Advanced system concepts for civil aircraft: An overview of avionic architectures. *Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering 209* (1995), 265—272.

[71] *Nejad, A. B., Molnos, A., and Goossens, K.* Enabling time-triggered scheduling on a composable embedded system. In *Annual Workshop on PROGram for Research on Embedded Systems & Software (PROGRESS)* (November 2011).

[72] *Nejad, A. B., Molnos, A., and Goossens, K.* A software-based technique enabling composable hierarchical preemptive scheduling for time-triggered applications. In *IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)* (August 2013), pp. 183–192.

[73] *Nejad, A. B., Molnos, A., and Goossens, K.* A unified execution model for multiple computation models of streaming applications on a composable MPSoC. *Journal of Systems Architecture - Embedded Systems Design 59*, 10-C (2013), 1032–1046.

[74] *Nejad, A. B., Molnos, A., and Goossens, K. G. W.* A unified execution model for data-driven applications on a composable MPSoC. In *Euromicro Conference on Digital System Design, Architectures, Methods and Tools (DSD)* (September 2011), pp. 818–822.

[75] *Nelson, A.* *Composable and Predictable Power Management.* PhD thesis, Delft University of Technology, The Netherlands, 2014.

[76] *Nelson, A., Molnos, A., and Goossens, K.* Composable power management with energy and power budgets per application. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)* (July 2011), pp. 396–403.

[77] *Nelson, A., Nejad, A. B., Molnos, A., Koedam, M., and Goossens, K.* CoMik: A predictable and cycle-accurately composable real-time microkernel. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)* (March 2014).

[78] *Nieuwland, A., Kang, J., Gangwal, O. P., Sethuraman, R., Busa, N., Goossens, K., Llopis, R. P., and Lippens, P.* 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*, 3 (2002), 233–270.

[79] *Parks, T. M., Pino, J. L., and Lee, E. A.* A comparison of synchronous and cyclestatic dataflow. In *The Twenty-Ninth Asilomar Conference on Signals, Systems & Computers* (1995), p. 204.

[80] *Philips Semiconductors. Device Transaction Level (DTL) Protocol Specification. Version 2.2*, July 2002.

[81] *Prisaznuk, P.* Arinc 653 role in integrated modular avionics (ima). In *Digital Avionics Systems Conference (DASC)* (Oct 2008).

[82] *Reyes, V., Bautista, T., Marrero, G., Carballo, P., and Kruijtzer, W.* CASSE: a system-level modeling and design-space exploration tool for multiprocessor systems-on-chip. In *Euromicro Conference on Digital System Design, Architectures, Methods and Tools (DSD)* (aug.-3 sept. 2004), pp. 476 – 483.

[83] *Samolej, S.* Arinc specification 653 based real-time software engineering. *e-Informatica 5*, 1 (2011), 39–49.

[84] *Sander, I., and Jantsch, A.* System modeling and transformational design refinement in ForSyDe. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 23*, 1 (jan. 2004), 17 – 32.

[85] *Sangiovanni-Vincentelli, A., and Natale, M. D.* Embedded system design for automotive applications. *The Computer journal 40*, 10 (2007), 42–51.

[86] *Schoeberl, M., Patel, H. D., and Lee, E. A.* Fun with a deadline instruction. Tech. rep., EECS Department, University of California, Berkeley, Oct 2009.

[87] *Shigero, S., Matsumoto, T., and Kei, H.* On the schedulability conditions on partial time slots. In *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)* (1999).

[88] *Shin, I., and Lee, I.* Periodic resource model for compositional real-time guarantees. In *Real-Time System Symposium (RTSS)* (2003).

[89] *Stefan, R. A., Molnos, A., and Goossens, K.* dAElite: A TDM NoC supporting QoS, multicast, and fast connection set-up. *IEEE Transactions on Computers 63*, 3 (2014), 583–594.

[90] *Stefanov, T., Zissulescu, C., Turjan, A., Kienhuis, B., and Deprettere, E.* System design using kahn process networks: The compaan/laura approach. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)* (Washington, DC, USA, 2004), IEEE Computer Society, p. 10340.

[91] *Stuijk, S. Predictable Mapping of Streaming Applications on Multiprocessors.* PhD thesis, Eindhoven University of Technology, The Netherlands, 2007.

[92] *Stuijk, S., and Basten, T.* Analyzing concurrency in streaming applications. *Journal of System Architecture 54*, 1-2 (Jan. 2008), 124–144.

[93] *Stuijk, S., Geilen, M., Theelen, B. D., and Basten, T.* Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)* (2011), pp. 404–411.

[94] *Šůcha, P., Kutil, M., Sojka, M., and Hanzálek, Z.* TORSCHE scheduling toolbox for Matlab. In *Computer Aided Control System Design (CACSD)* (2006).

[95] *Thiele, L., and Wilhelm, R.* Design for timing predictability. *Real-Time Systems 28*, 2-3 (Nov. 2004), 157–177.

[96] *Tokar, J. L.* Space & time partitioning with ARINC 653 and pragma profile. In *International Real-Time Ada Workshop (IRTAW)* (2003).

[97] *Turjan, R., and Kienhuis, B.* Translating affine nested-loop programs to process networks. In *In Proceedings of the international conference on Compilers, architecture, and synthesis for embedded systems (CASES)* (2004), ACM Press, pp. 220–229.

[98] *van den Heuvel, M. M. H. P., Holenderski, M., Cools, W., Bril, R. J., and Lukkien, J. J.* Virtual timers in hierarchical real-time systems. In *Real-Time System Symposium (RTSS) – Work in Progress Session* (2009).

[99] *Wiggers, M., Bekooij, M., Jansen, P. G., and Smit, G. J. M.* Efficient computation of buffer capacities for multi-rate real-time systems with back-pressure. In *IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)* (2006).

[100] Xilinx, Inc. `http://www.xilinx.com/`.

[101] *Xilinx, Inc.* *MicroBlaze Processor Reference Guide*, 2012.

[102] *Yang, J., Kim, H., Park, S., Hong, C., and Shin, I.* Implementation of compositional scheduling framework on virtualization. *SIGBED Rev. 8*, 1 (Mar. 2011).

[103] *Zimmer, M., Broman, D., Shaver, C., and Lee, E. A.* Flexpret: A processor platform for mixed-criticality systems. Tech. rep., EECS Department, University of California, Berkeley, Oct 2013.

[104] *Zissulescu, C., Stefanov, T., Kienhuis, B., and Deprettere, E. F.* Laura: Leiden architecture research and exploration tool. In *International Conference on Field Programmable Logic and Applications (FPL)* (2003), pp. 911–920.

[105] *Zou, J., Matic, S., Lee, E., Feng, T., and Derler, P.* Execution strategies for PTIDES, a programming model for distributed embedded systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)* (april 2009), pp. 77 –86.

List of Publications

International Journals

[1] Kees Goossens, Arnaldo Azevedo, Karthik Chandrasekar, Manil Dev Gomony, Sven Goossens, Martijn Koedam, Yonghui Li, Davit Mirzoyan, Anca Molnos, **Ashkan Beyranvand Nejad**, Andrew Nelson, and Shubhendu Sinha. **Virtual execution platforms for mixed-time-criticality systems: the CompSOC architecture and design flow**. *SIGBED Review*, 10(3):23–34, 2013.

[2] **Ashkan Beyranvand Nejad**, Anca Molnos, and Kees Goossens. **A unified execution model for multiple computation models of streaming applications on a composable MPSoC**. *Journal of Systems Architecture - Embedded Systems Design*, 59(10-C):1032–1046, 2013.

[3] **Ashkan Beyranvand Nejad**, Anca Molnos, Matias Escudero Martinez, and Kees Goossens. **A hardware/software platform for QoS bridging over multi-chip NoC-based systems**. *Parallel Computing*, 39(9):424–441, 2013.

International Conferences

[4] Kees Goossens, Bart Vermeulen, and **Ashkan Beyranvand Nejad**. **A high-level debug environment for communication-centric debug**. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 202–207, April 2009.

[5] Sven Goossens, Benny Akesson, Martijn Koedam, **Ashkan Beyranvand Nejad**, Andrew Nelson, and Kees Goossens. **The CompSOC design flow for virtual execution platforms**. In *10th FPGAworld Conference*, September 2013.

[6] Andrew Nelson, **Ashkan Beyranvand Nejad**, Anca Molnos, Martijn Koedam, and Kees Goossens. **CoMik: A predictable and cycle-accurately composable real-time microkernel**. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, March 2014.

[7] Shubhendu Sinha, Martijn Koedam, Rob van Wijk, Andrew Nelson, **Ashkan Beyranvand Nejad**, Marc Geilen, and Kees Goossens. **Composable and predictable dynamic loading for time-critical partitioned systems**. In *Euromicro Symposium on Digital System Design (DSD)*, August 2014.

[8] **Ashkan Beyranvand Nejad**, Matias Escudero Martinez, and Kees Goossens. **An FPGA bridge preserving traffic quality of service for on-chip network-based systems**. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 425–430, March 2011.

[9] **Ashkan Beyranvand Nejad**, Anca Molnos, and Kees G. W. Goossens. **A Unified Execution Model for Data-Driven Applications on a Composable MPSoC**. In *14th Euromicro Conference on Digital System Design, Architectures, Methods and Tools (DSD)*, pages 818–822, September 2011.

[10] **Ashkan Beyranvand Nejad**, Anca Molnos, and Kees Goossens. **A software-based technique enabling composable hierarchical preemptive scheduling for time-triggered applications**. In *IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 183–192, August 2013.

National & International Workshops

[11] Benny Akesson, Er Stuijk, Anca Molnos, Martijn Koedam, Radu Stefan, Andrew Nelson, **Ashkan Beyranvand Nejad**, and Kees Goossens. **Virtual Platforms for Mixed Time-Criticality Applications: The CoMPSoC Architecture and SDF3 Design Flow**. March 2012.

[12] Anca Molnos, **Ashkan Beyranvand Nejad**, Ba Thang Nguyen, Sorin Cotofana, and Kees Goossens. **Decoupled Inter- and Intra-application Scheduling for Composable and Robust Embedded MPSoC Platforms**. In *5th Workshop on Mapping of Applications to MPSoCs & 15th International Workshop on Software and Compilers for Embedded Systems (Map2MPSoC/SCOPES 2012)*, pages 13–21, May 2012.

[13] Andrew Nelson, **Ashkan Beyranvand Nejad**, Davit Mirzoyan, Sorin Cotofana, and Kees Goossens. **Embedded Computer Architecture Laboratory: A Hands-on Experience Programming Embedded Systems with Resource and Energy Constraints**. In *Workshop on Embedded and Cyber-Physical Systems Education (WESE)*, October 2012.

[14] Radu Stefan, **Ashkan Beyranvand Nejad**, and Kees Goossens. **Online Allocation for Contention-free-routing NoCs**. In *Interconnection Network Architecture: On-Chip, Multi-Chip Workshop (INA-OCMC)*, pages 13–16, January 2012.

[15] **Ashkan Beyranvand Nejad**, Kees Goossens, Johan Walters, and Bart Kienhuis. **Mapping KPN Models of Streaming Applications on A Network-on-Chip Platform**. In *Annual Workshop on Circuits, Systems and Signal Processing (ProRISC)*, November 2009.

[16] **Ashkan Beyranvand Nejad**, Matias Escudero Martinez, and Kees Goossens. **On-Chip Interconnect Protocol Stack Exploration for FPGA Board-to-Board Bridging**. In *Annual Workshop on PROGram for Research on Embedded Systems & Software (PROGRESS)*, November 2010.

[17] **Ashkan Beyranvand Nejad**, Anca Molnos, and Kees Goossens. **Enabling Time-Triggered Scheduling on a Composable Embedded System**. In *Annual Workshop on PROGram for Research on Embedded Systems & Software (PROGRESS)*, November 2011.

Samenvatting

R*ECENTE* ontwikkelingen laten een sterke trend zien om meerdere applicaties te integreren in één embedded systeem. Hierbij worden Multi-Processor System-on-Chip architecturen geopperd als de oplossing voor het integreren van complexe ontwerpen op een embedded systeem. Dit doel wordt gehaald door zo veel mogelijk reken resources te integreren in een enkele chip en daarmee de parallelle executie van meerdere applicaties te ondersteunen. Vanwege kostenoverwegingen moeten applicaties resources delen om parallelle executie mogelijk te maken op deze systemen.

Om de volledige rekenkracht van een MPSoC te benutten worden applicaties gesplitst in meerdere parallelle taken. Afhankelijk van het intrinsieke gedrag van de applicatie zijn deze taken echter data- of tijdsafhankelijk van elkaar. Voor het implementeren van deze applicaties worden er twee verschillende execution models gebruikt, namelijk data of tijd gedreven. Hiernaast hebben deze applicaties vaak timing eisen, deze zijn in drie groepen te classificeren: firm, soft en non real-time. In een *mixed time-criticality* systeem stellen de verschillende applicaties zeer uiteenlopende eisen aan de timing en als ze resources delen zal dit interferentie tussen de applicaties veroorzaken.

Om het uitvoeren van verschillende real-time applicaties op een embedded systeem mogelijk te maken moet het gedrag van het systeem voorspelbaar zijn. Op deze manier kan er gegarandeerd worden dat de timing eisen van iedere applicatie worden gehaald. Om bovendien onafhankelijke ontwikkeling, verificatie en integratie mogelijk te maken van *mixed-criticality* applicaties moet het systeem sterk composable zijn. In andere woorden, de applicaties zijn volledig geïsoleerd in tijd. Het tijd gedrag van iedere applicatie is onafhankelijk op klok-slag niveau. Op deze manier is tijdsinterferentie tussen de applicaties volledig voorkomen.

In deze dissertatie behandelen wij twee belangrijke uitdagingen bij het ontwerpen en prototyping van *mixed time-criticality* systemen: (i) Implementeren van sterk composable Virtual Platforms voor mixed-criticality embedded systemen, en (ii) stellen wij een uniforme abstracte executielaag voor, voor het uitvoeren van applicaties met verschillende models of execution. Hierbij richten wij ons op tijd-gedreven modellen en Kahn Process Network en dataflow (met name Cyclo-Static DataFlow), als de twee varianten van data gedreven models of computation. Op basis van deze uitdagingen beantwoorden wij de hoofd onderzoeksvraag: *hoe ontwerp en voer je meerdere applicaties tegelijkertijd uit op een embedded systeem, gegeven dat de applicaties gebruik maken van verschillende*

models of computation en niveaus van time-criticality?

Deze dissertatie stelt als oplossing voor om voor iedere applicatie een Virtual Platform te creëren dat iedere hardware resource, die bij de uitvoering van de applicatie is betrokken, virtualiseert. Hierom is een temporale partitioneringstechniek toegepast op de CompSOC hardware architectuur. Bovenop deze architectuur is de CoMik microkernel ontworpen als een minimale softwarelaag die resource partitionering mogelijk maakt. CoMik maakt, controleert en schedules processor partities, en voert applicaties uit binnen hun applicatiepartities door de virtualizatie van de processor zijn software hooks en biedt aan iedere applicatie een Application Programming Interface aan voor het gebruik van de gevirtualizeerde resources. Hierdoor kunnen applicaties direct op het virtuele platform uitgevoerd worden, identiek aan hoe ze op het dedicated hardware platform uitgevoerd zouden worden. Er is echter een executie abstractiegat tussen de semantiek van de models of computation en de door het platform aangeboden primitieve operaties. Om dit gat te overbruggen is een model of execution geïntroduceerd om een algemene set executie operaties te definiëren en hoe met deze operaties een specifiek model of computation is te implementeren.

Het model of execution is geïmplementeerd in de vorm van een lichtgewicht besturingssysteem-bibliotheek, genaamd CompOSe, die uitgevoerd word in iedere partitie van het VP. CompOSe is ontworpen als een set softwarecomponenten en de implementatie introduceert geen onvoorspelbaar gedrag bij de uitvoering van een applicatie en behoudt de composability eigenschappen van het systeem.

We demonstreren dat onze voorgestelde technieken het mogelijk maken meerdere applicaties, geïmplementeerd met verschillende models of execution composable en voorspelbaar tegelijkertijd uit te voeren doormiddel van twee experimentele opstellingen: (i) Een matlab simulatie omgeving voor het onderzoeken van het temporale gedrag van de CoMik microkernel. (ii) Een Field Programmable Gate Array (FPGA) prototype van het CompSOC platform wordt gebruikt voor het bestuderen van de composability eigenschappen en het ondersteunen van meerdere models of computation door het CompOSe real-time besturingssysteem.

STELLINGEN

behorende bij het proefschrift

Composable Virtual Platforms for Mixed-Criticality Embedded Systems

van

Ashkan Beyranvand Nejad

1. In een mixed time-criticality systeem, is de compexiteit van ontwerp, verificatie en integratie van applicaties grotendeels een gevolg van niet-kritische applicaties. [dit proefschrift]
2. Een executiemodel is nodig om de ruimte in executie abstractie te overbruggen tussen het model van de berekeningsemantiek en de primitieve executie werkzaamheden van het onderliggende executie platform. [dit proefschrift]
3. Voorspelbaarheid is een bottom-up eigenschap van een system, wat inhoudt dat het niet mogelijk is om een voorspelbaar systeem te hebben, dat is ontwikkeld met onvoorspelbare componenten. [dit proefschrift]
4. De meeste verschijnselen die bekend staan als willekeurig, zijn intrinsiek zeer voorspelbaar, en het is het negeren, abstraheren van of het gebrek aan begrip voor de oorzaken dat ze willekeurig of onvoorspelbaar lijken.
5. Een ideale dictatuur presteert beter dan een ideale democratie , maar de ergste democratie presteert beter dan de ergste dictatuur.
6. Een academische graad, zoals een promotie, weerspiegelt een set (academische) vaardigheden die zijn verworven door de eigenaar aan een universiteit, in plaats van dat het wijst op een kennisniveau van de eigenaar (kandidaat) tijdens de verdediging.
7. Blinde beoordeling, enkel- of dubbelvoudig, is geen eerlijk mechanisme om een wetenschappelijk werk te beoordelen.
8. Niet alle meningen or overtuigingen verdienen respect, maar het recht on ze te hebben moet worden gerespecteerd.
9. Een van de grootste paradoxen in de technologie is om domme computer systemen ’smart devices’ te noemen.
10. De wereld is een gevaarlijke plaats om te wonen, niet vanwege de mensen die kwaad willen, maar vanwege de mensen die er niets tegen doen. [Albert Einstein]

Deze stellingen worden opponeerbaar en verdedigbaar geacht en zijn als zodanig goedgekeurd door de promotor Prof.dr. K.G.W. Goossens.

PROPOSITIONS

belonging to the thesis

Composable Virtual Platforms for Mixed-Criticality Embedded Systems

by

Ashkan Beyranvand Nejad

1. In a mixed time-criticality system, the complexity of designing, verifying, and integrating applications is largely due to the non-critical applications. [This thesis]
2. A model of execution is necessary to fill the gap of execution abstraction between the model of computation's semantics and the primitive execution operations of the underlying execution platform. [This thesis]
3. Predictability is a bottom-up property of a system, which means that it is not possible to have a predictable system developed by using unpredictable components. [This thesis]
4. Most of phenomena known as to be random (unpredictable) are very predictable intrinsically, and it is just ignoring, abstracting from, or lacking the understanding of the causes that makes them to seem random or unpredictable.
5. An ideal dictatorship outperforms an ideal democracy, but the worst democracy outperforms the worst dictatorship.
6. An academic degree, such as a PhD, reflects that a set of (academic) skills have been acquired by its owner at a university rather than indicating a level of knowledge of the owner at the time of the graduation.
7. Blind reviewing, either single or double, is not a fair mechanism to judge a scientific work.
8. Not all opinions or beliefs deserve respect, but the right of having them must be respected.
9. One of the biggest paradoxes in the technology is to call the stupid computer systems smart devices.
10. The world is a dangerous place to live, not because of the people who are evil, but because of the people who do not do anything about it. [Albert Einstein]

These propositions are regarded as opposable and defendable, and have been approved as such by the promotor Prof.dr. K.G.W. Goossens.

About the Author

Ashkan Beyranvand Nejad was born in Tehran, Iran, in 1983. He received his B.Sc. degree in Electrical and Electronics Engineering from Iran University of Science and Technology (IUST), in 2005, and his M.Sc. degree in Systems-on-Chip (SoC) Design from Royal Institute of Technology (KTH), Stockholm, Sweden, in 2008. In the last year of his M.Sc. study, he moved to the Netherlands to carry out his thesis on communication-centric transaction-based debug of SoCs in NXP Semiconductors research group (formerly Philips Research), in Eindhoven. Since 2009, he started with the faculty of Electrical Engineering, Mathematics and Computer Science, at Delft University of Technology, to pursue his PhD at Computer Engineering (CE) laboratory on the major topic of design challenges in mixed-criticality embedded systems. The outcomes of his PhD work have contributed to two main projects: (i) Tera-Scale Multi-core Processor ARchitecture (TSAR), a European funded project on design and application of multi-core processor architectures targeting tera-flops performance, and (ii) CompSOC research platform developed by Eindhoven University of Technology in collaboration with Delft University of Technology. His current research interests include composable and predictable embedded Systems-on-Chip, mixed-criticality systems, real-time resource management, execution models, and on-chip interconnect architectures.

TU Delft Delft University of Technology

CE

ISBN 9789461863768

9 789461 863768